cs5460/6460: Operating Systems

Lecture: System Init (Kernel Memory Allocator and Page Table)

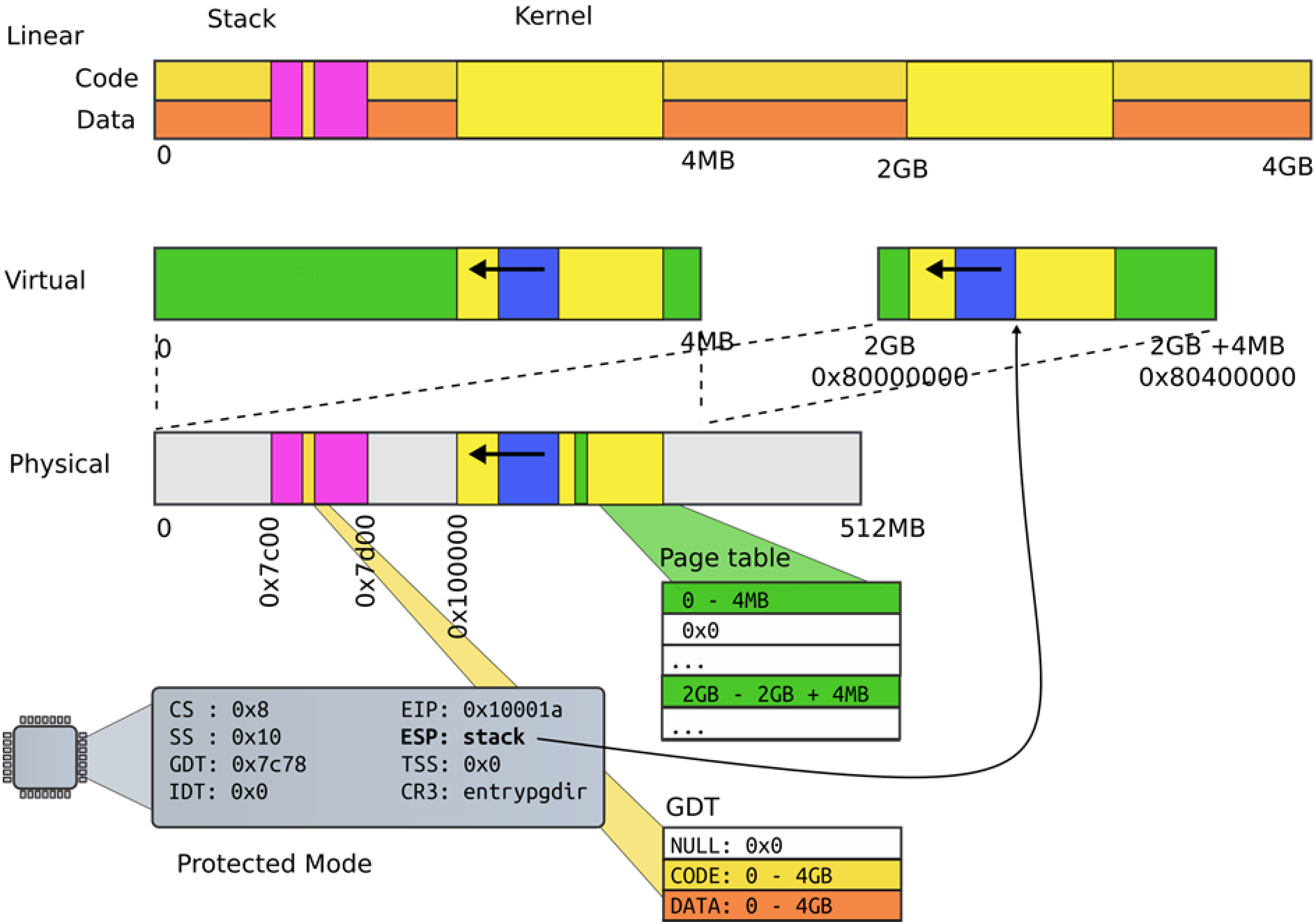
Anton Burtsev

February, 2025

Recap of the boot sequence

- Setup segments (data and code)
- Switched to protected mode
- Loaded GDT (segmentation is on)
- Setup stack (to call C functions)
- Loaded kernel from disk
- Setup first page table
  - 2 entries [0 : 4MB] and [2GB : (2GB + 4MB)]
- Setup high-address stack
- Jumped to main()

State of the system after boot

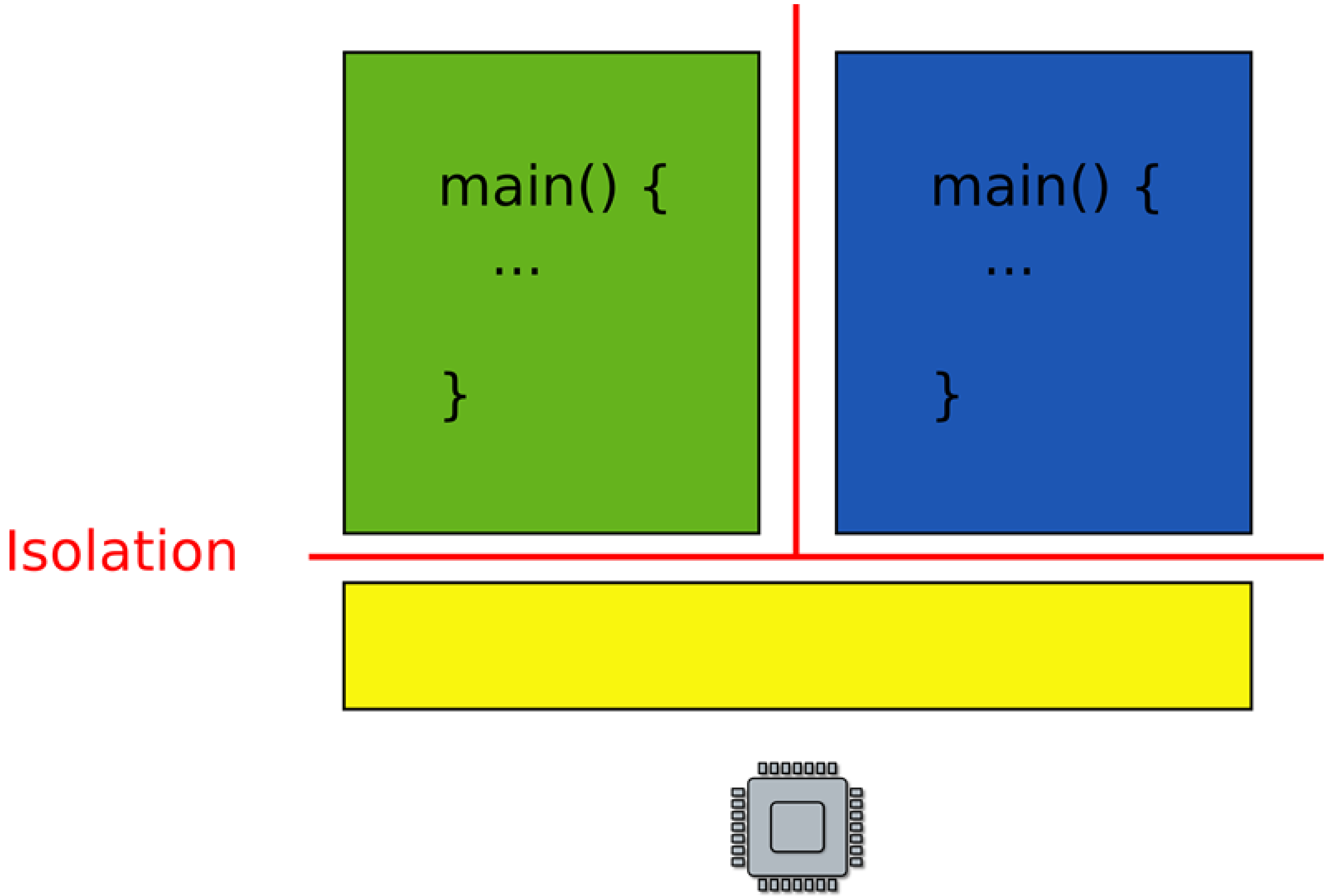


Running in main()

```
// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
    lapicinit(); // interrupt controller
    seginit(); // segment descriptors
    cprintf("\ncpu%d: starting xv6\n\n", cpunum());
...
}
```

What's next?

We want to run multiple processes



But what is a process?

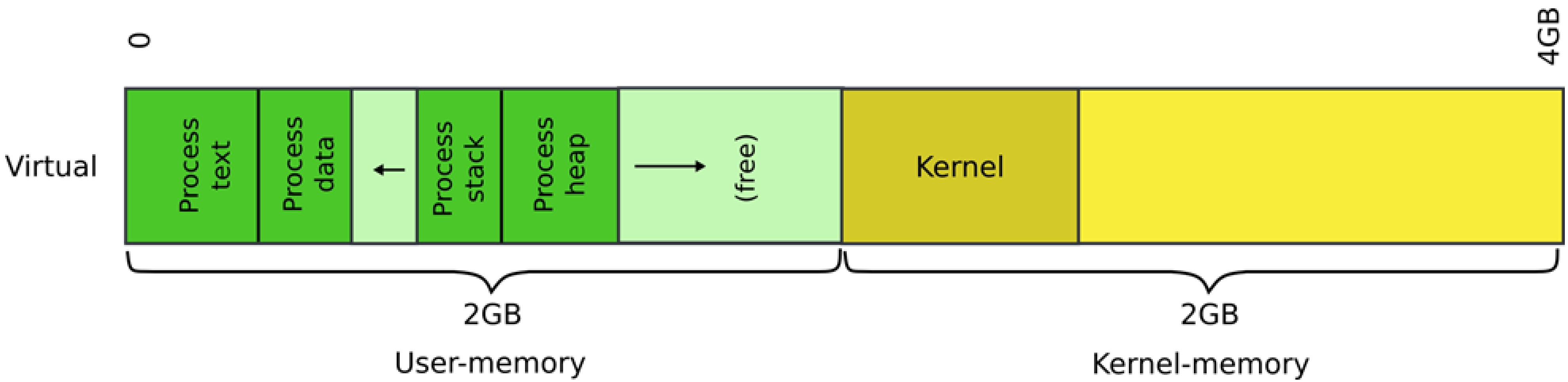
A couple of requirements

- Each process is a collection of resources
- Memory
  - E.g., text, stack, heap
- In-kernel state
  - E.g., open file descriptors, network sockets (connections)

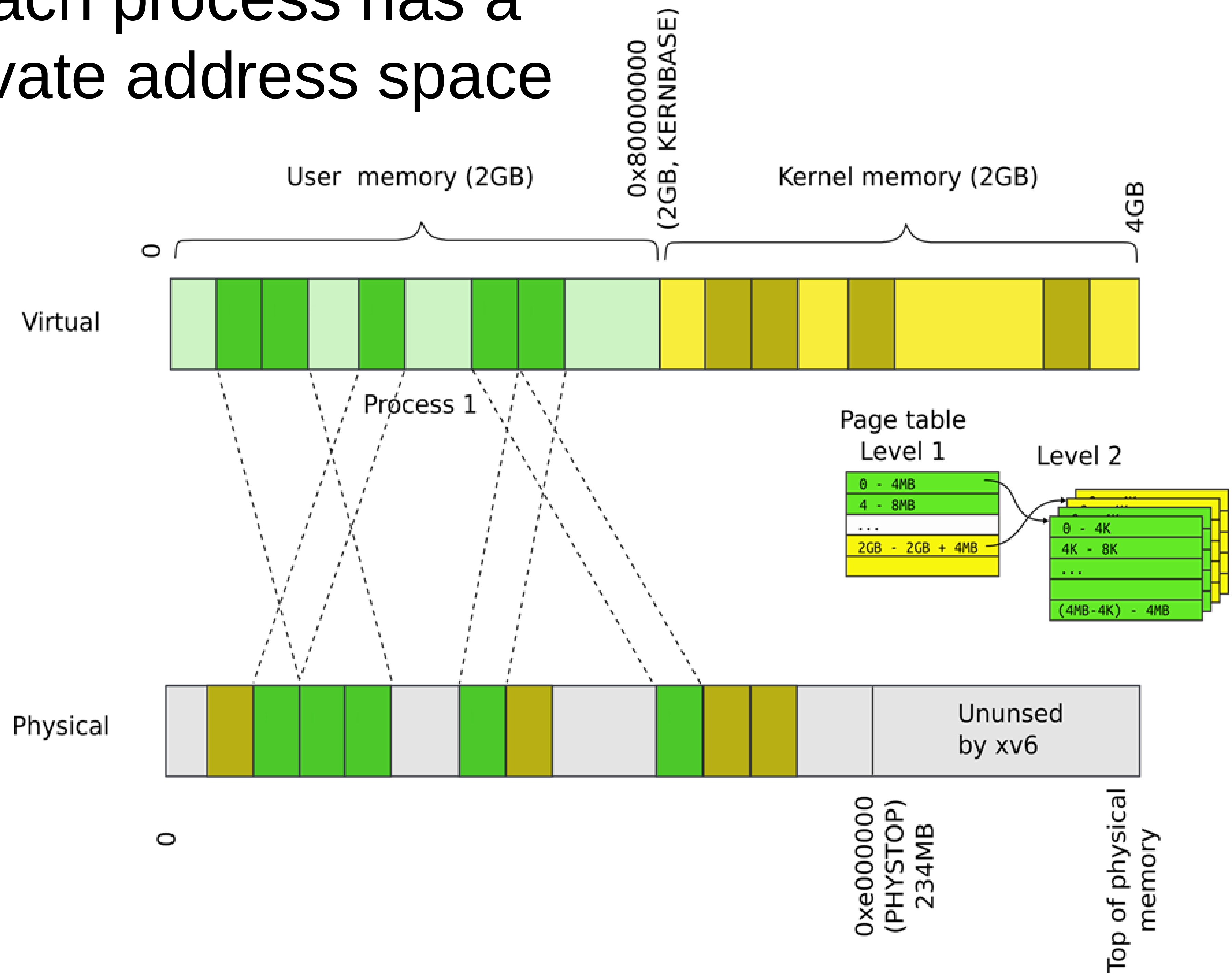
A couple of requirements

- Each process is a collection of resources
- Memory
  - E.g., text, stack, heap
- In-kernel state
  - E.g., open file descriptors, network sockets (connections)
- Processes are isolated from each other
- Processes don't trust each other
  - Individual users, some privileged
- Can't interfere with other processes
- Can't change kernel (to affect other processes)

Each process will have a 2GB/2GB private address space

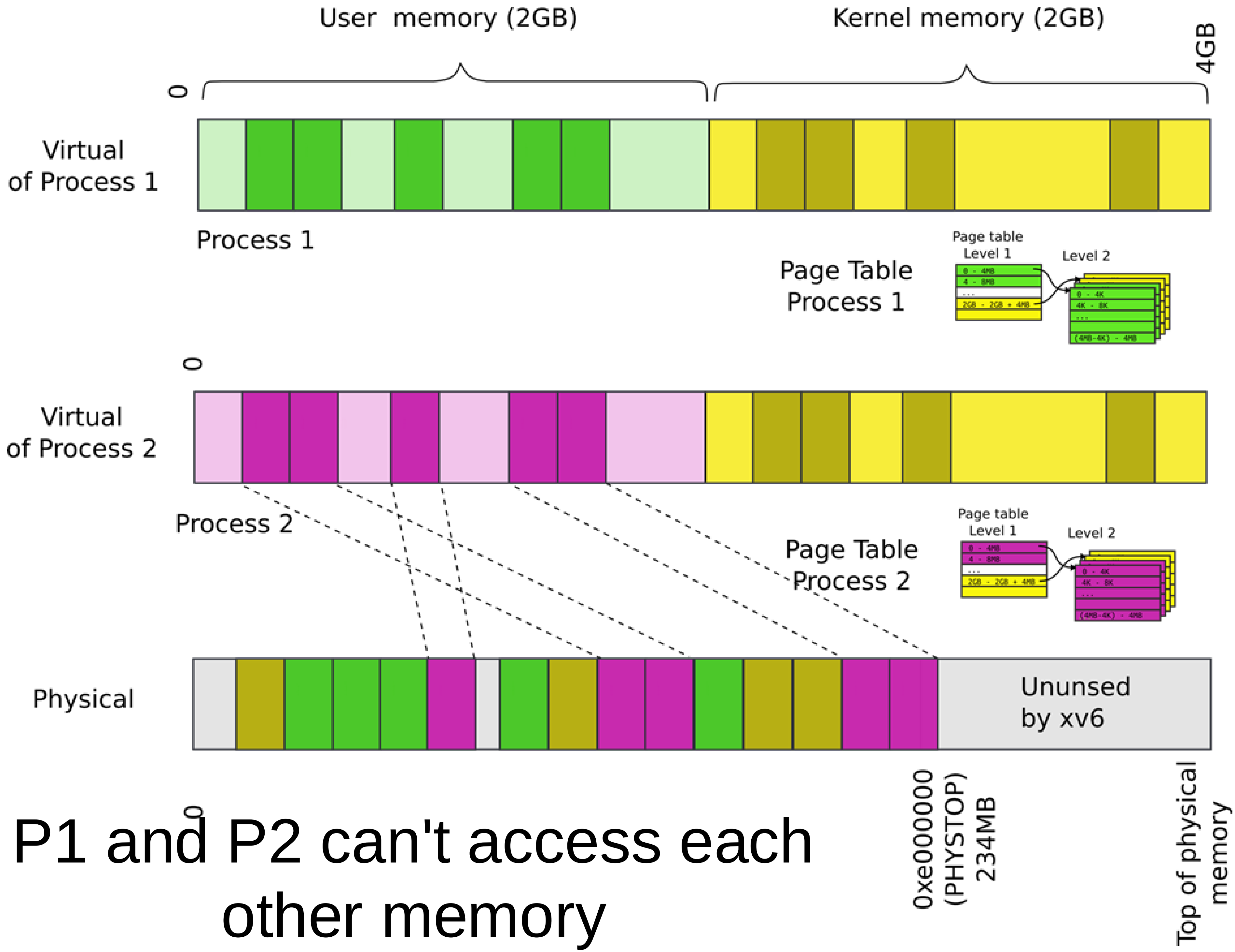


Each process has a private address space



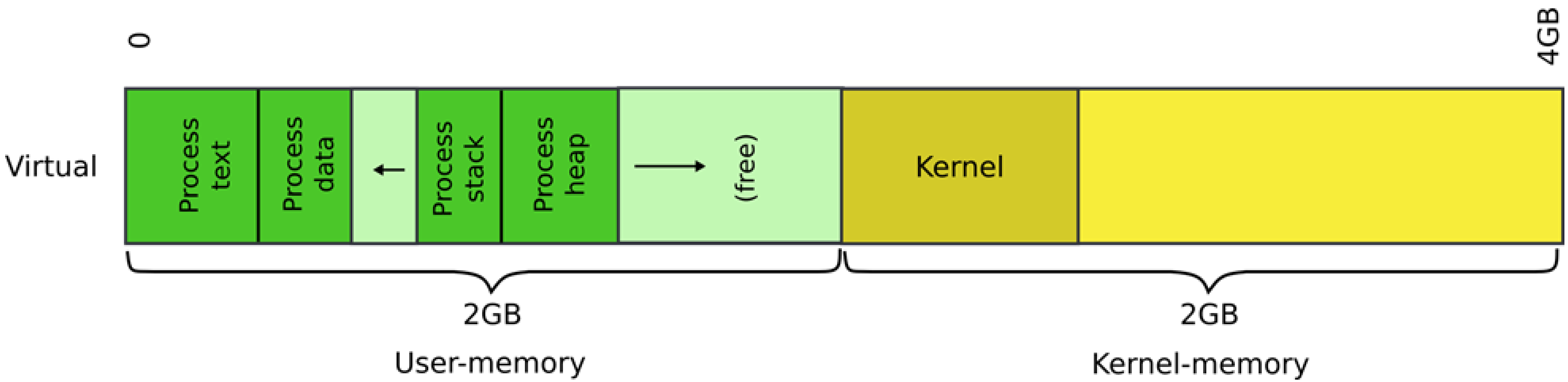
Each process maps the kernel

- It's not strictly required
- But convenient for system calls
- No need to change the page table when process enters the kernel with a system call
- **Things are much faster!**

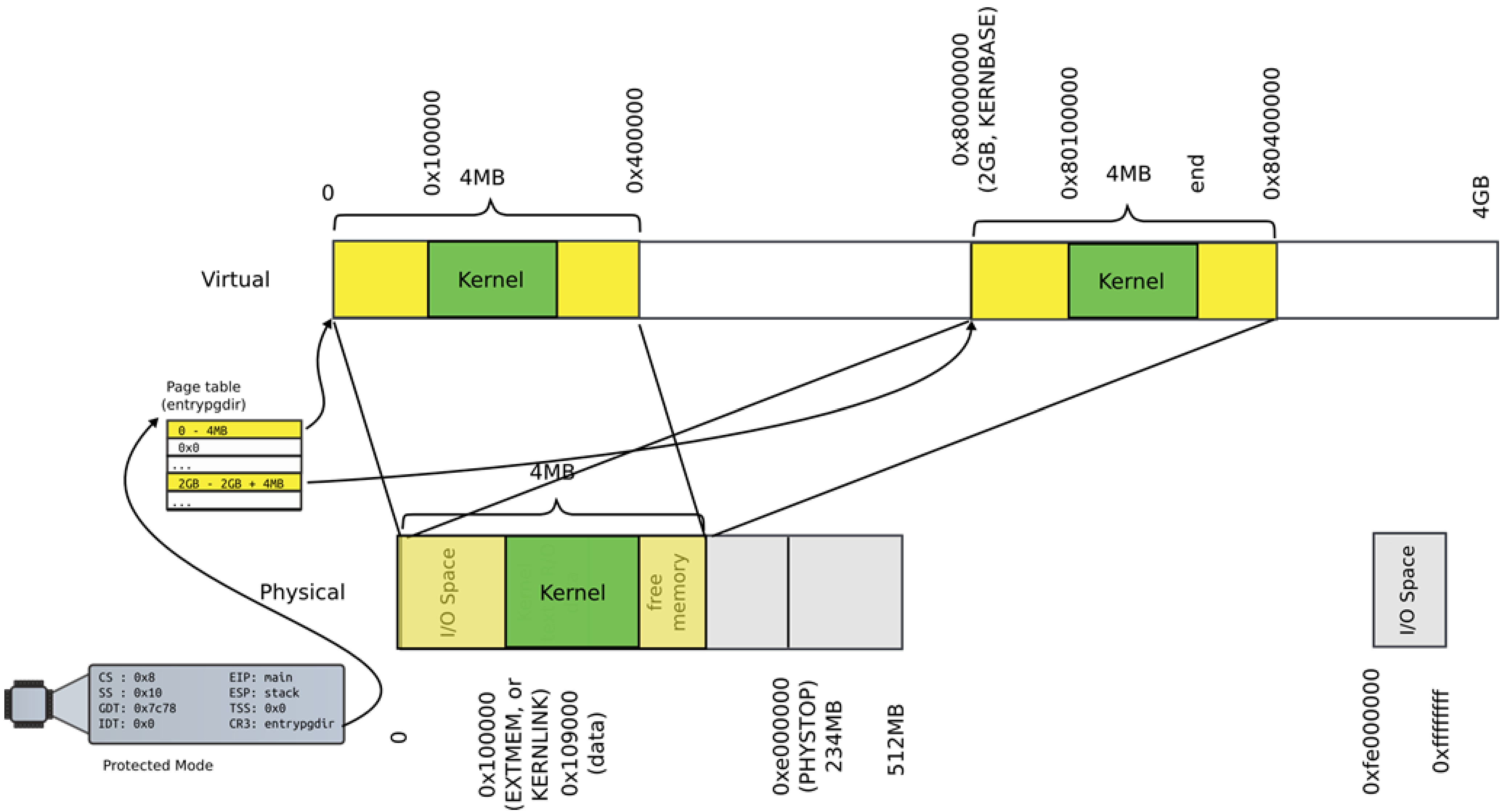


P1 and P2 can't access each other memory

Our goal: 2GB/2GB address space



Memory after boot



Outline

- Create the kernel address space
- Create kernel memory allocator
- Allocate memory for page tables
  - Page table directory and page table

Kernel memory allocator

- Kernel needs normal 2 level, 4KB page table
- Right now we have
  - One (statically allocated) page table
  - That has only two entries
- And it is a page table for 4MB pages
- 4KB page table is a better choice
  - Xv6 processes are small
  - Wasting 4MB on a program that fits into 1KB is absurd

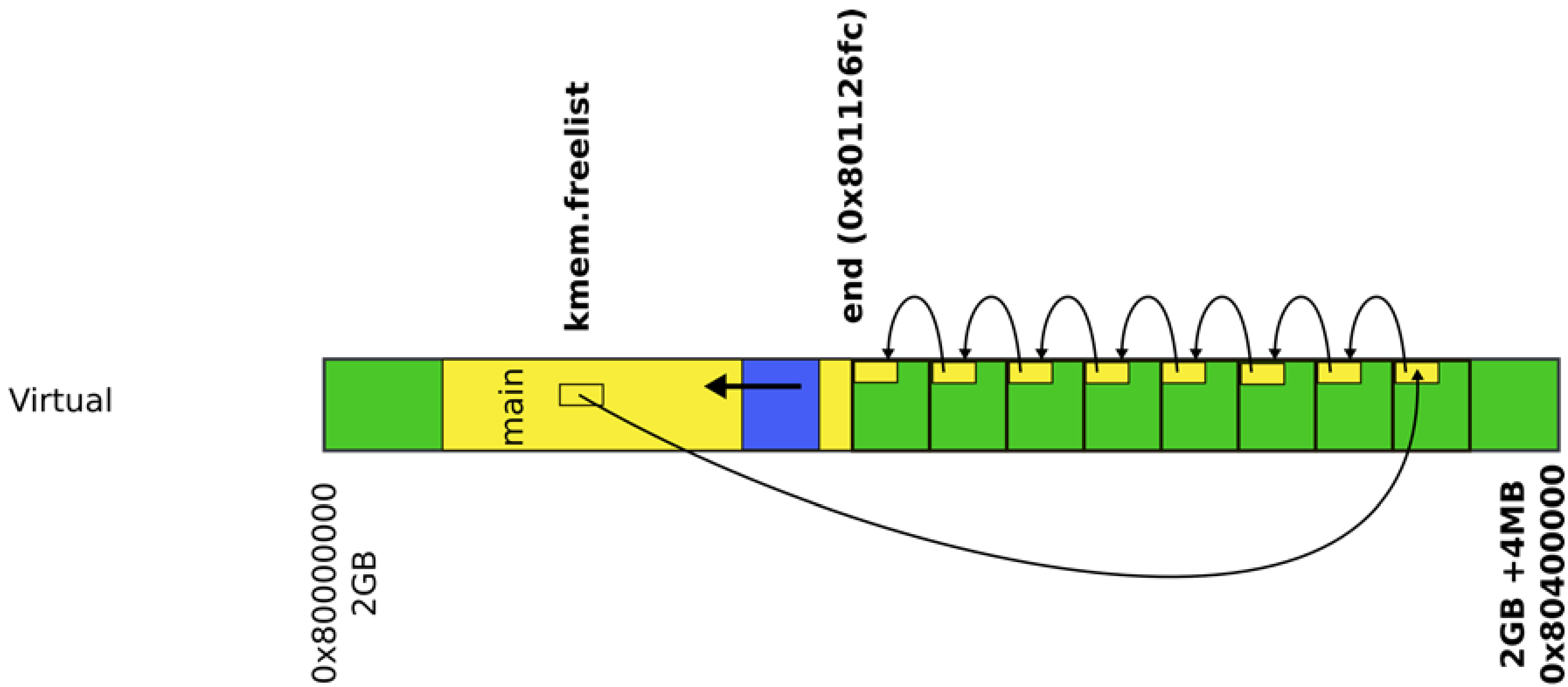
- But to create page tables we need memory
- Where can it come from?

Simple memory allocator

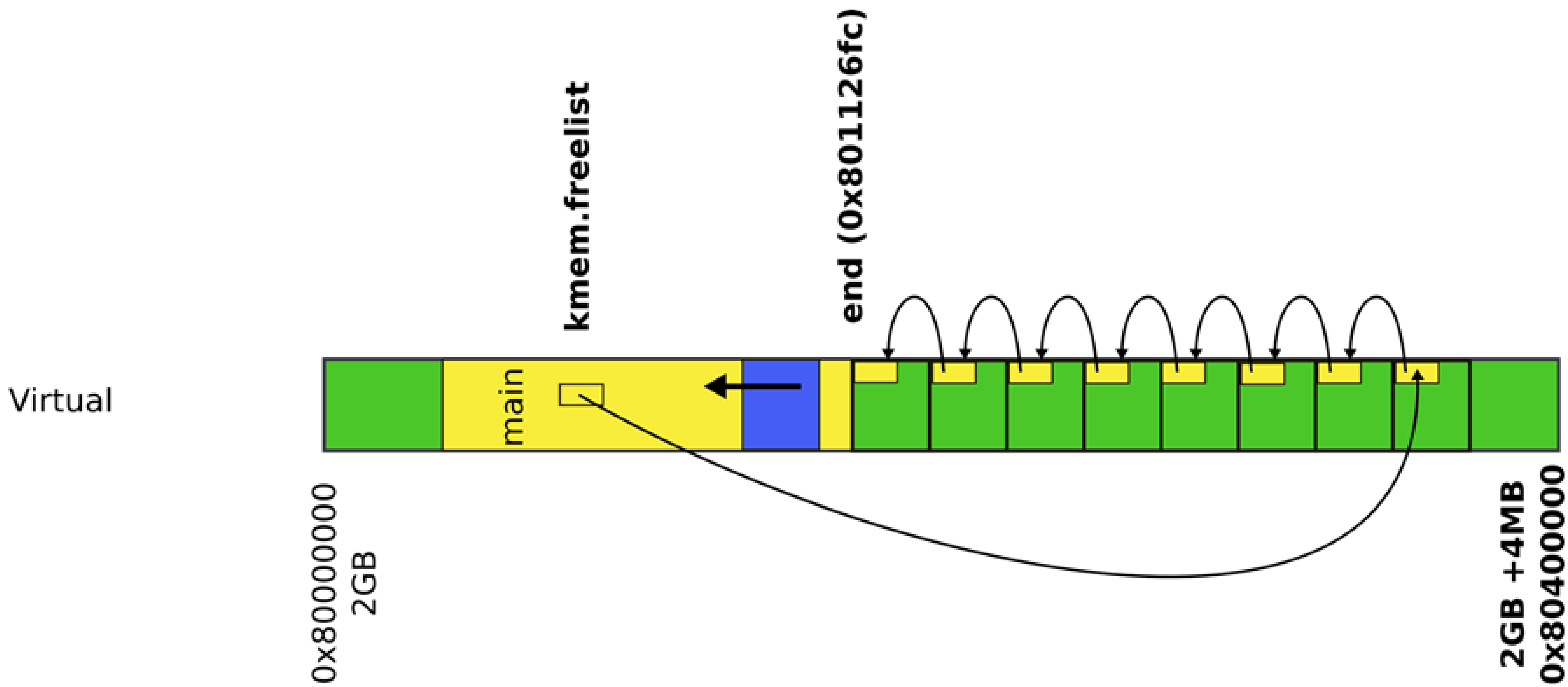
- Goal:
  - alloc() and free()
  - To allocate page tables, stacks, data structures, etc.

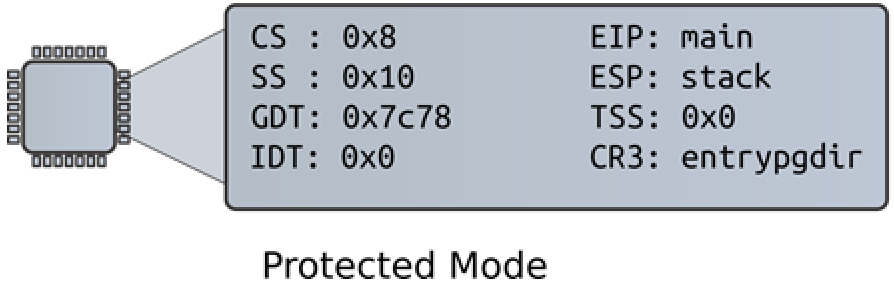
What can it look like?

Page allocator



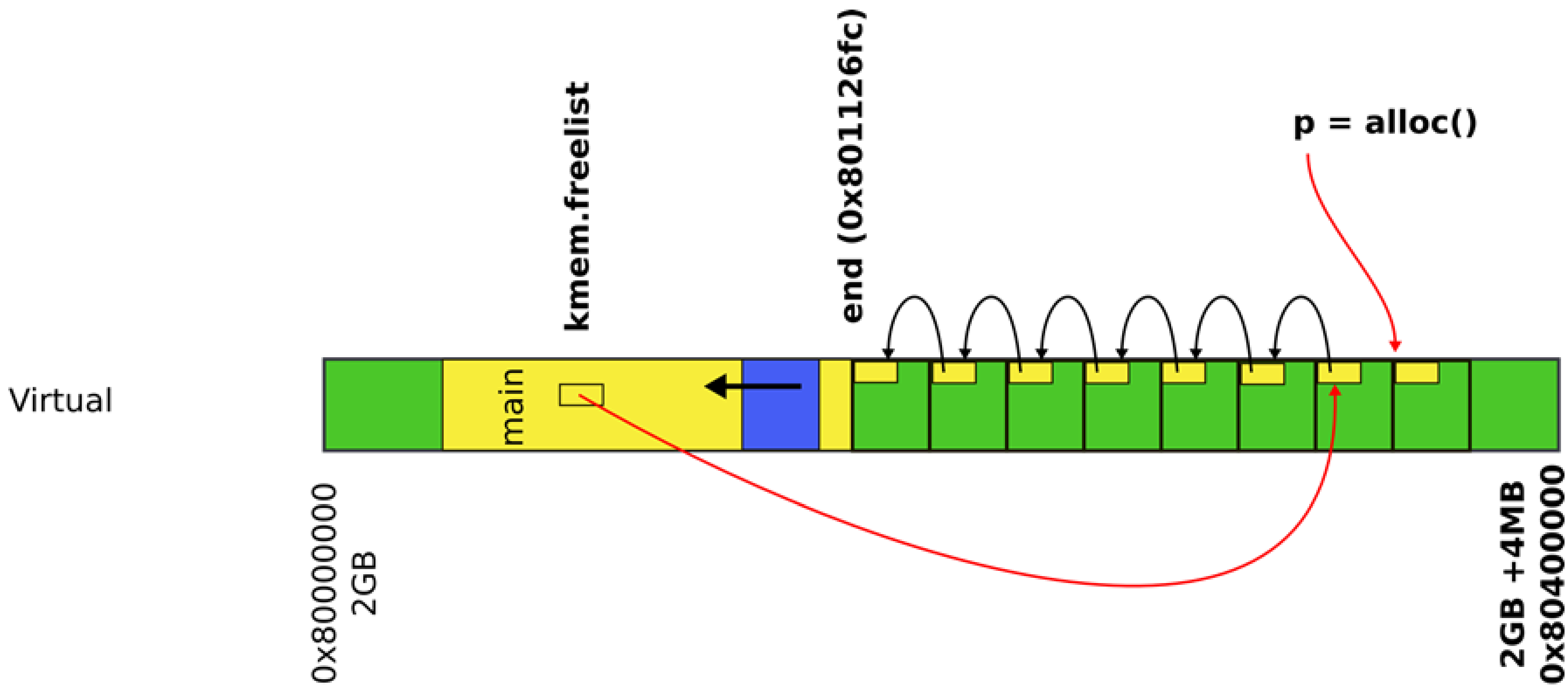
Page allocator





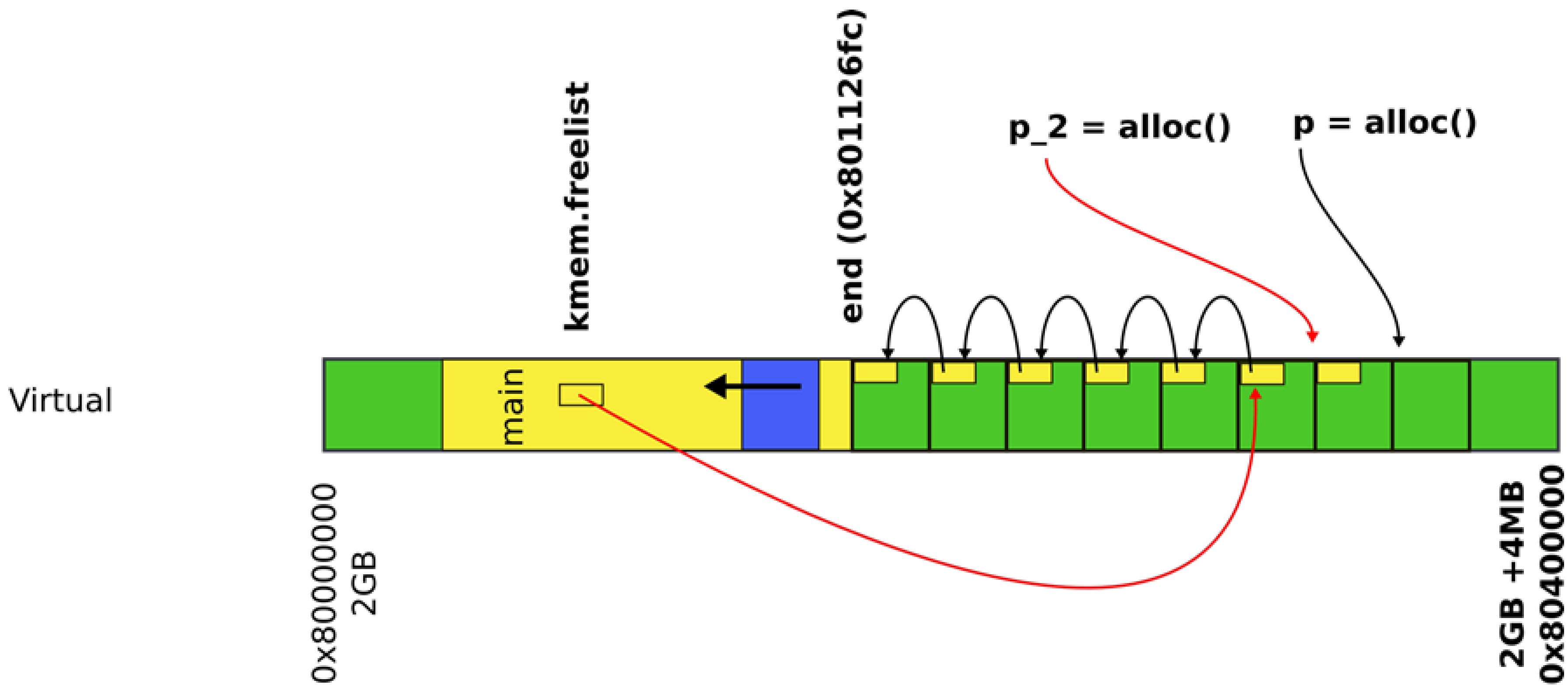

```
3014 struct run {
3015   struct run *next;
3016 };
```

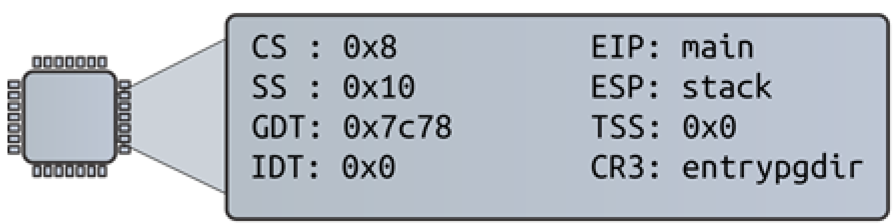
Page allocator




```
3014 struct run {
3015   struct run *next;
3016 };
```

Page allocator

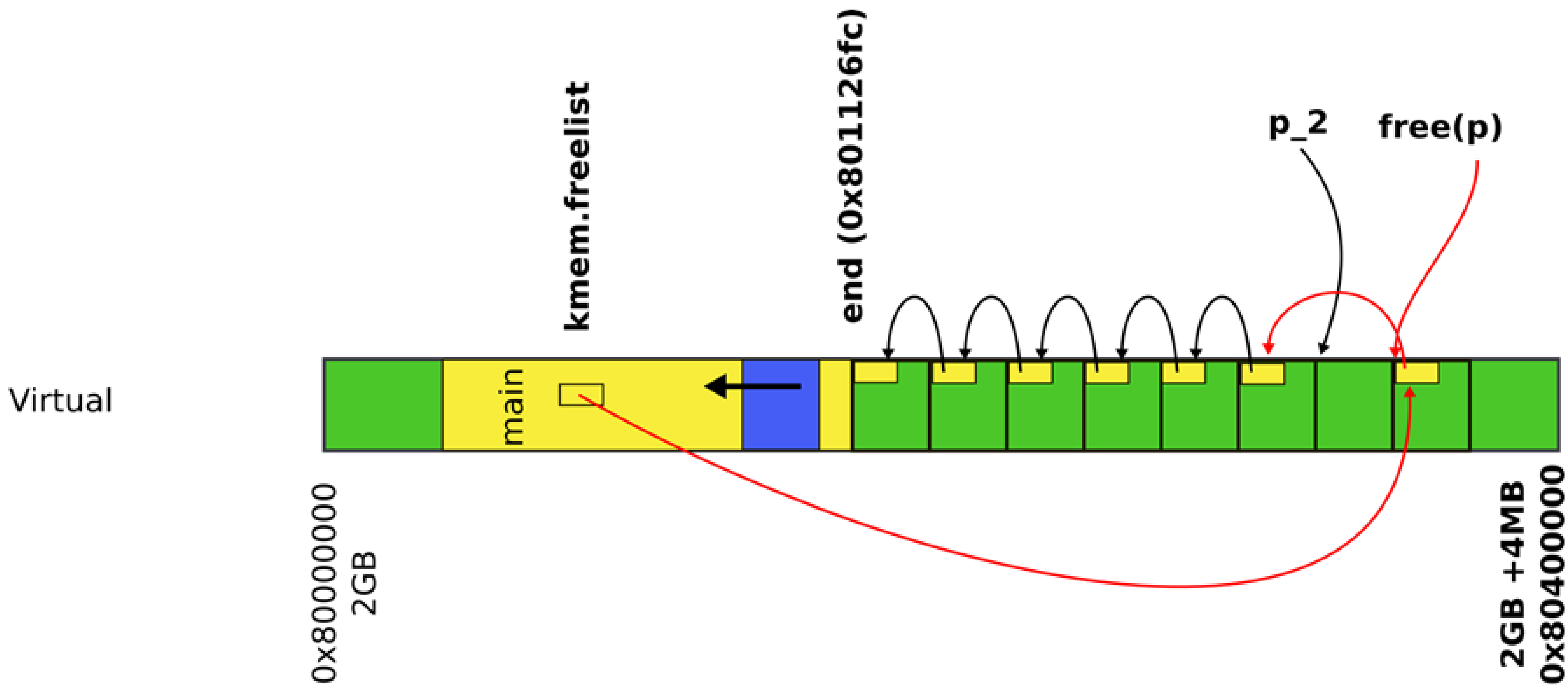


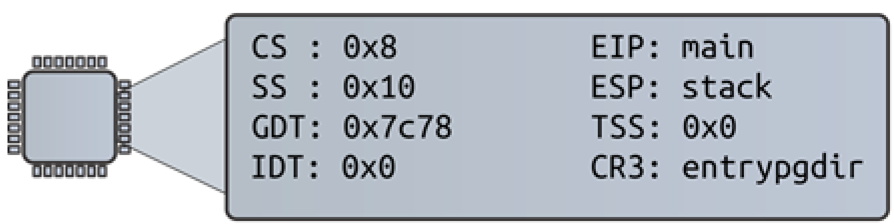


Protected Mode

```
3014 struct run {
3015   struct run *next;
3016 };
```

Page allocator



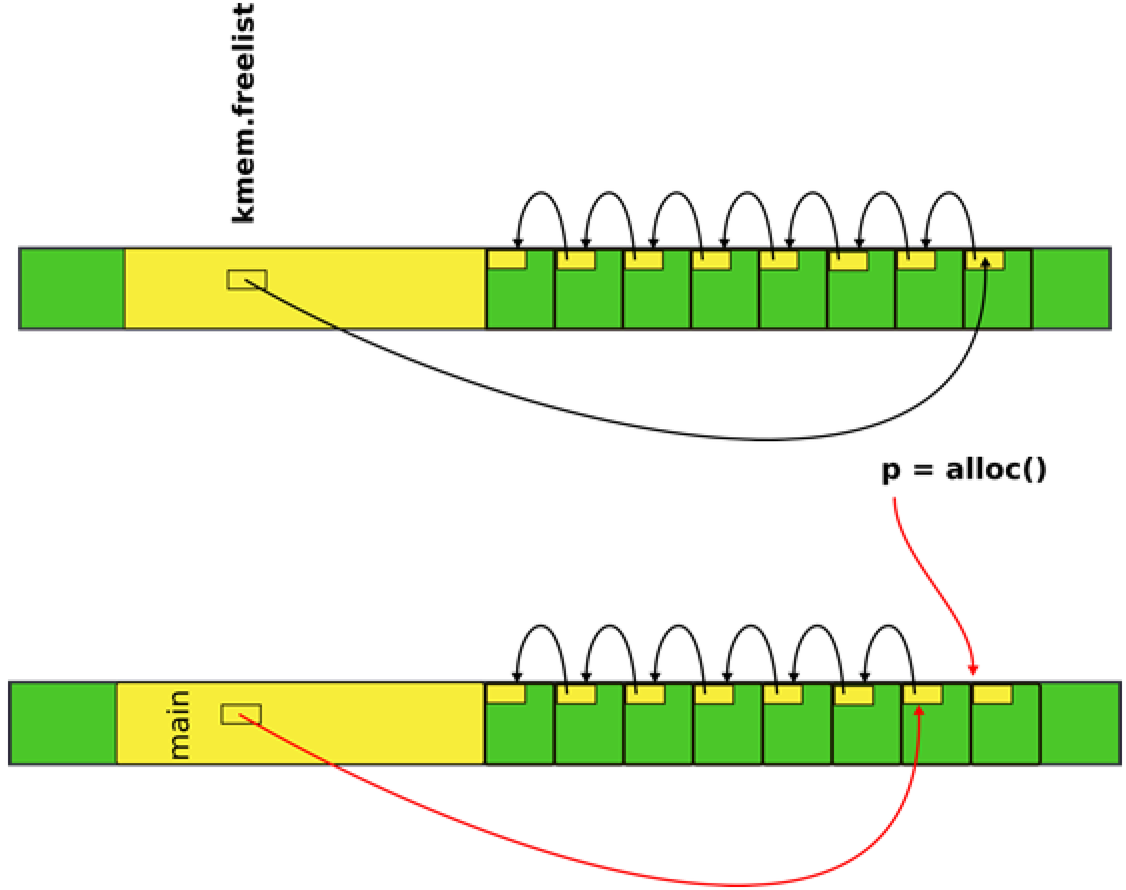


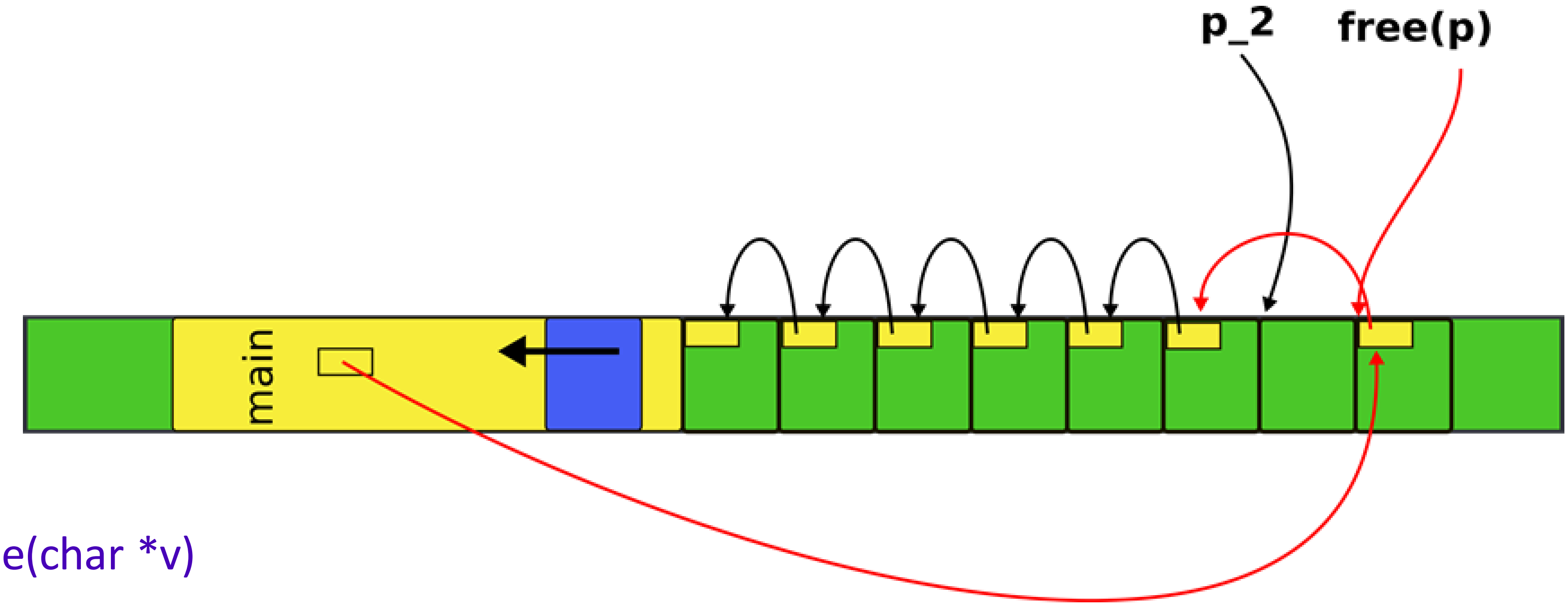
Protected Mode

```
3014 struct run {
3015   struct run *next;
3016 };
```

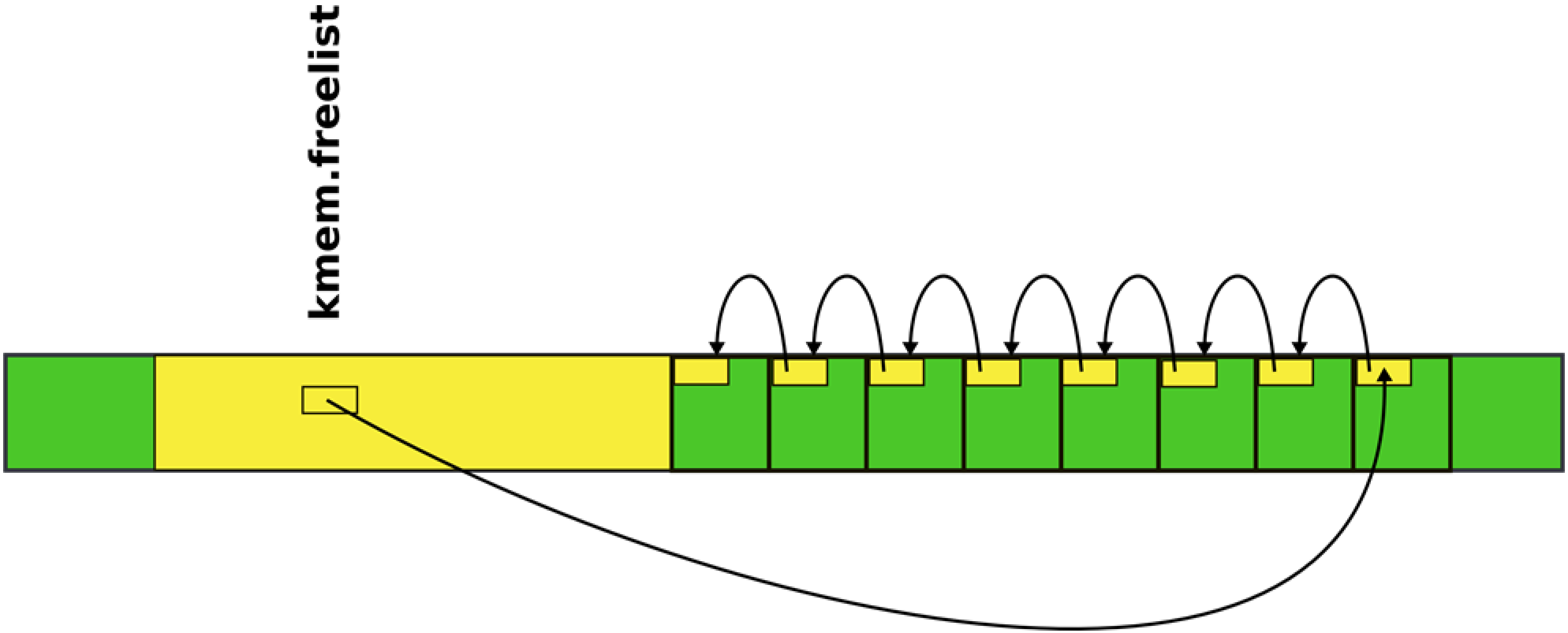
kalloc() - kernel allocator

```
3087 char*
3088 kalloc(void)
3089 {
3080   struct run *r;
...
3094   r = kmem.freelist;
3095   if(r)
3096     kmem.freelist = r->next;
...
3099   return (char*)r;
3099 }
```



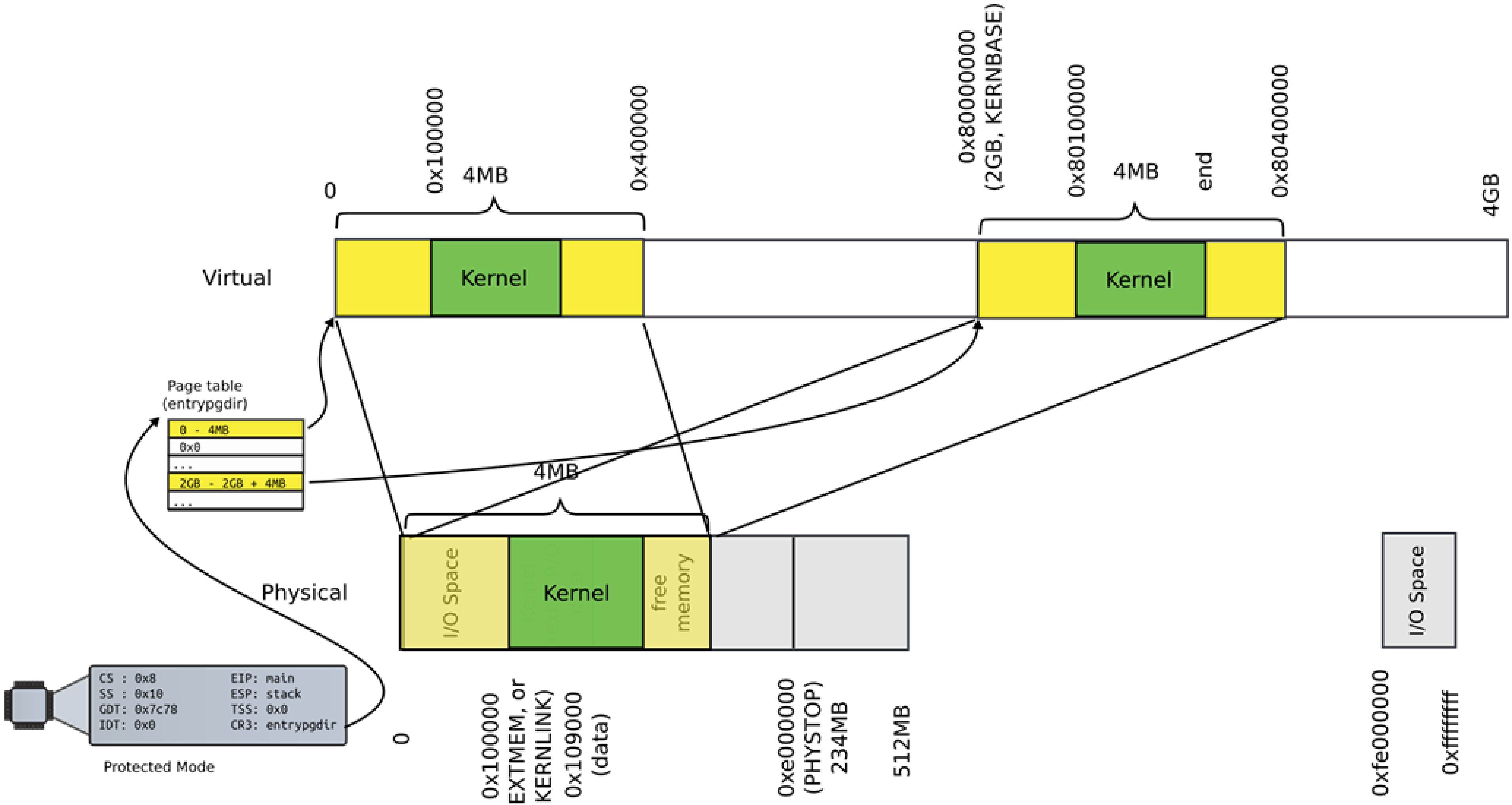



```
3065 kfree(char *v)
3066 {
3067   struct run *r;
...
3077   r = (struct run*)v;
3078   r−>next = kmem.freelist;
3079   kmem.freelist = r;
...
2832 }
```

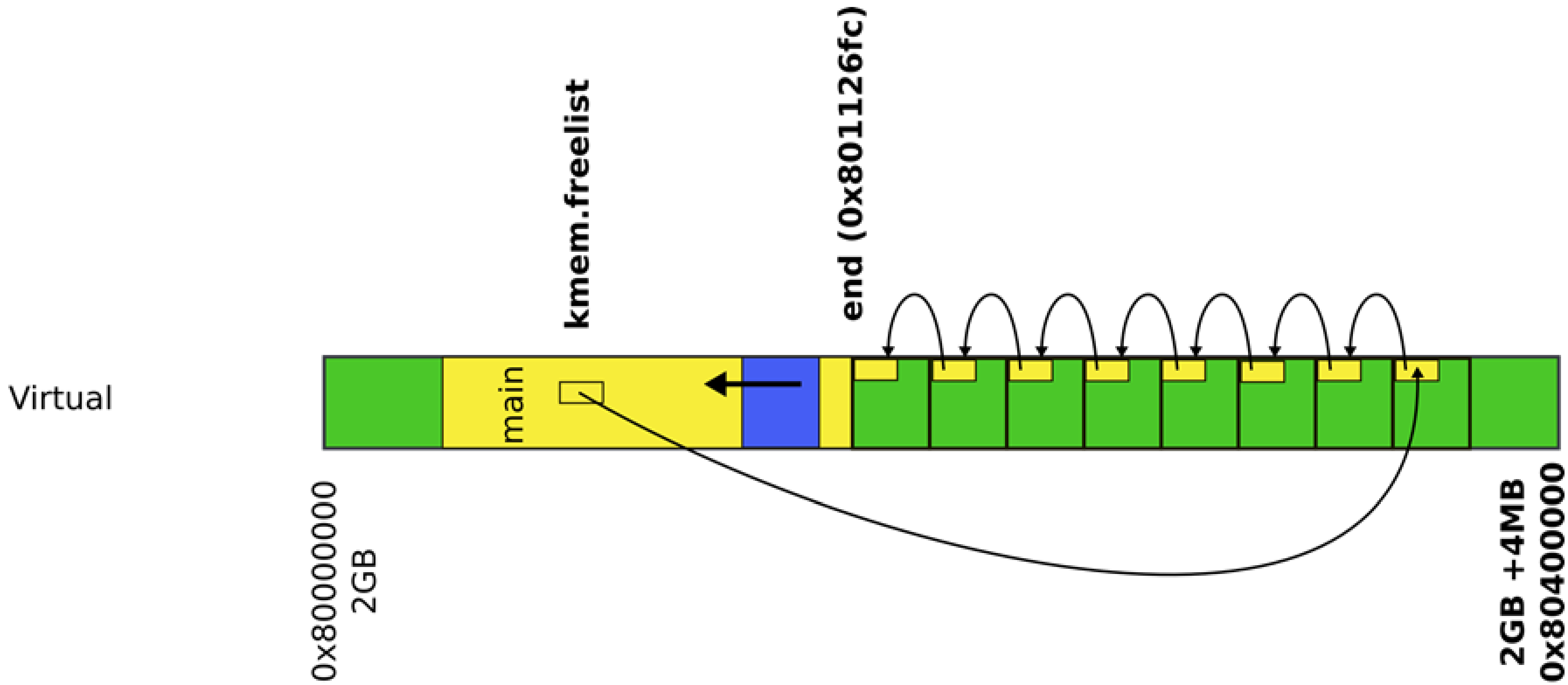


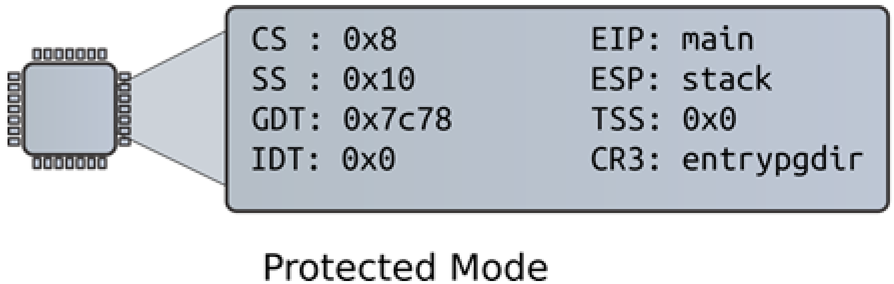
- Where can we get memory to keep the list itself?
- Note, the list is maintained within each page
- It has to write each page though to update the "next" pointer

There is a bit of free memory in the 4MB page we've mapped



Donate this free memory to the allocator





- Take memory from the end of the kernel binary
- To the end of the 4MB page

kinit1(): initialize the allocator with free memory

```
int
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
    lapicinit(); // interrupt controller
    seginit(); // segment descriptors
    cprintf("\ncpu%d: starting xv6\n\n", cpunum());
    picinit(); // another interrupt controller
    ioapicinit(); // another interrupt controller
    consoleinit(); // console hardware
    uartinit(); // serial port
...
}
```

Freerange()

```
kinit1(void *vstart, void *vend)
{
...
  freerange(vstart, vend);
}
```

- Free range of memory from vstart to vend giving it to the allocator
- i.e., adding pages to the list

freerange()

```
freerange(void *vstart, void *vend)
{
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
    kfree(p);
}
```

- freerange() internally simply frees the pages from vstart to vend
- kfree() adds them to the allocator list

Where do we start?

```
1316 int
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
```

- What is this **end**?

```
1311 extern char end[];
```

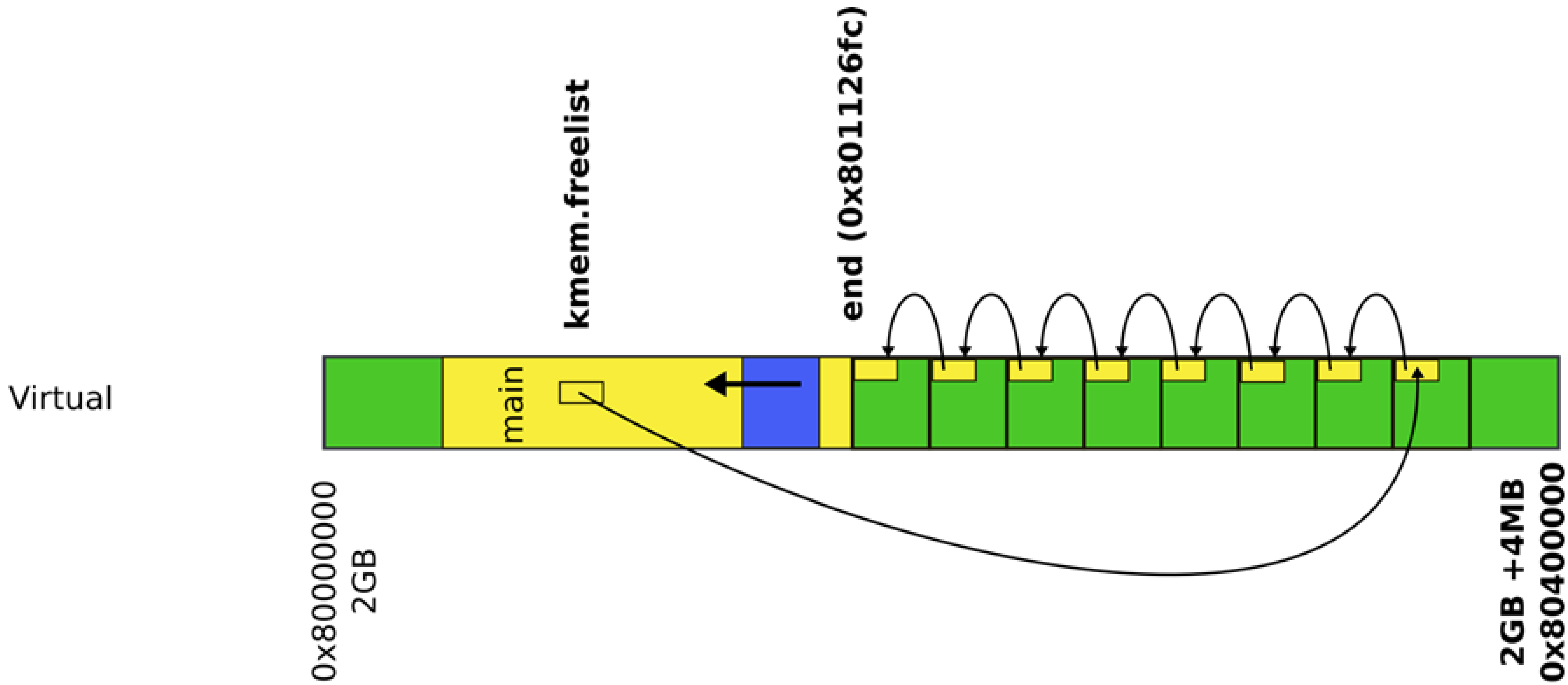
Where do we start?

```
int
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
```

- What is this **end**?

```
extern char end[]; // first address after
                      kernel loaded from ELF file
```

Donate this free memory to the allocator



Recap

- Kernel has a memory allocator
- It allocates memory in chunks of 4KB
- Good enough to maintain kernel data structures

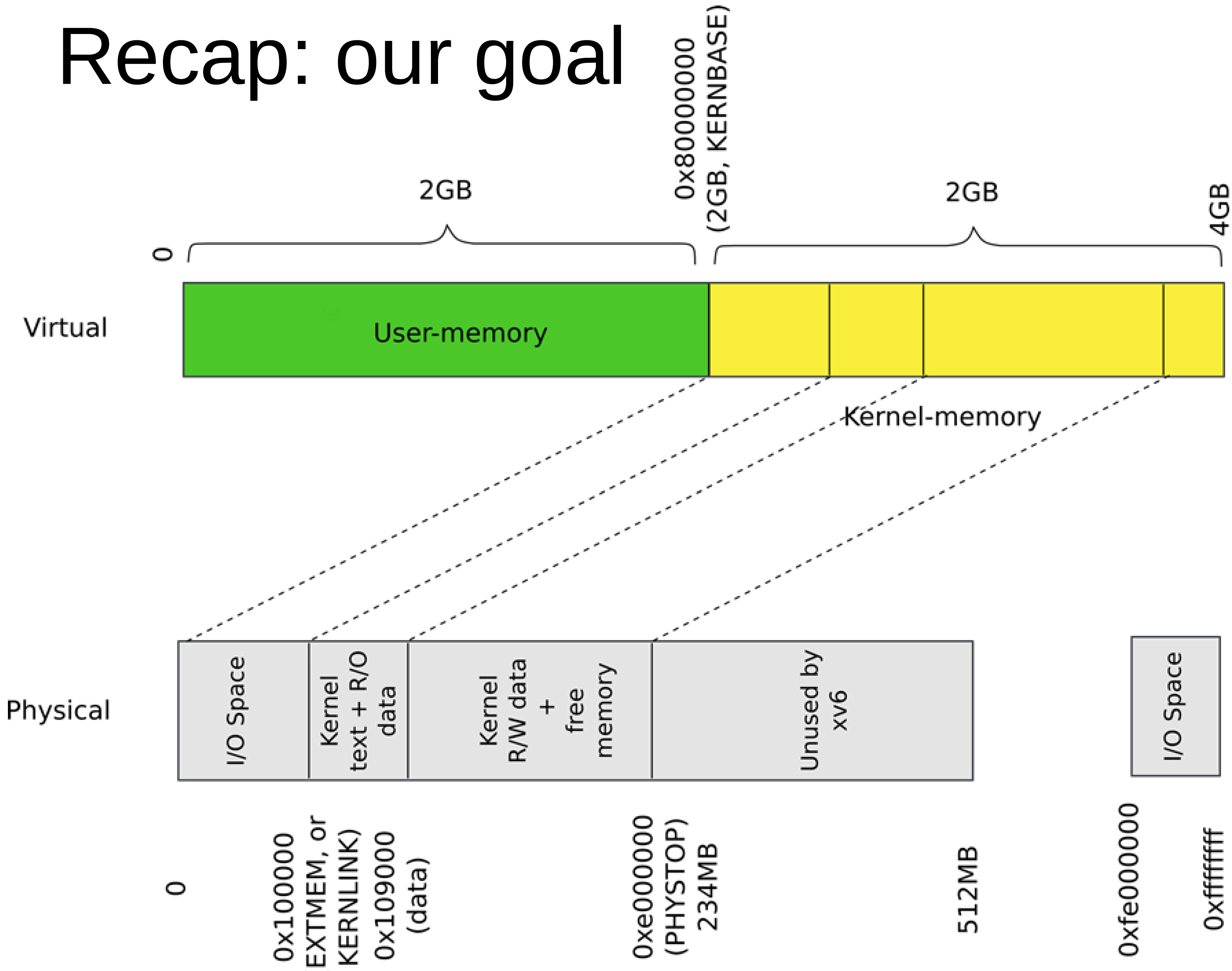
Kernel page table
(for 4KB page tables)

Back to main(): Kernel address space

```
int
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
    lapicinit(); // interrupt controller
    seginit(); // segment descriptors
    cprintf("\ncpu%d: starting xv6\n\n", cpunum());
    picinit(); // another interrupt controller
    ioapicinit(); // another interrupt controller
    consoleinit(); // console hardware
    uartinit(); // serial port
...
}
```

- What do you think has to happen?
  - i.e., how do we construct a kernel page table?

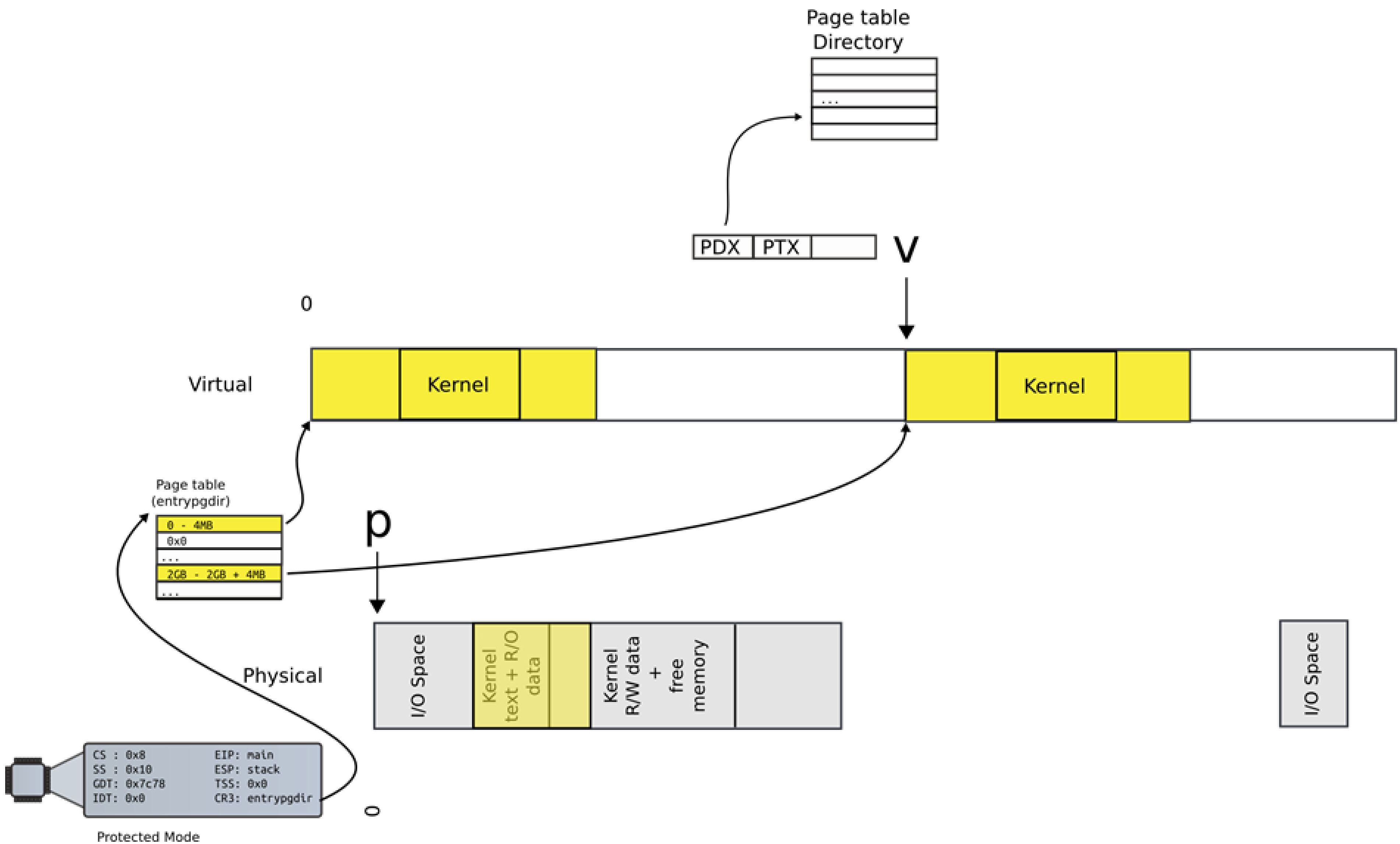
Recap: our goal



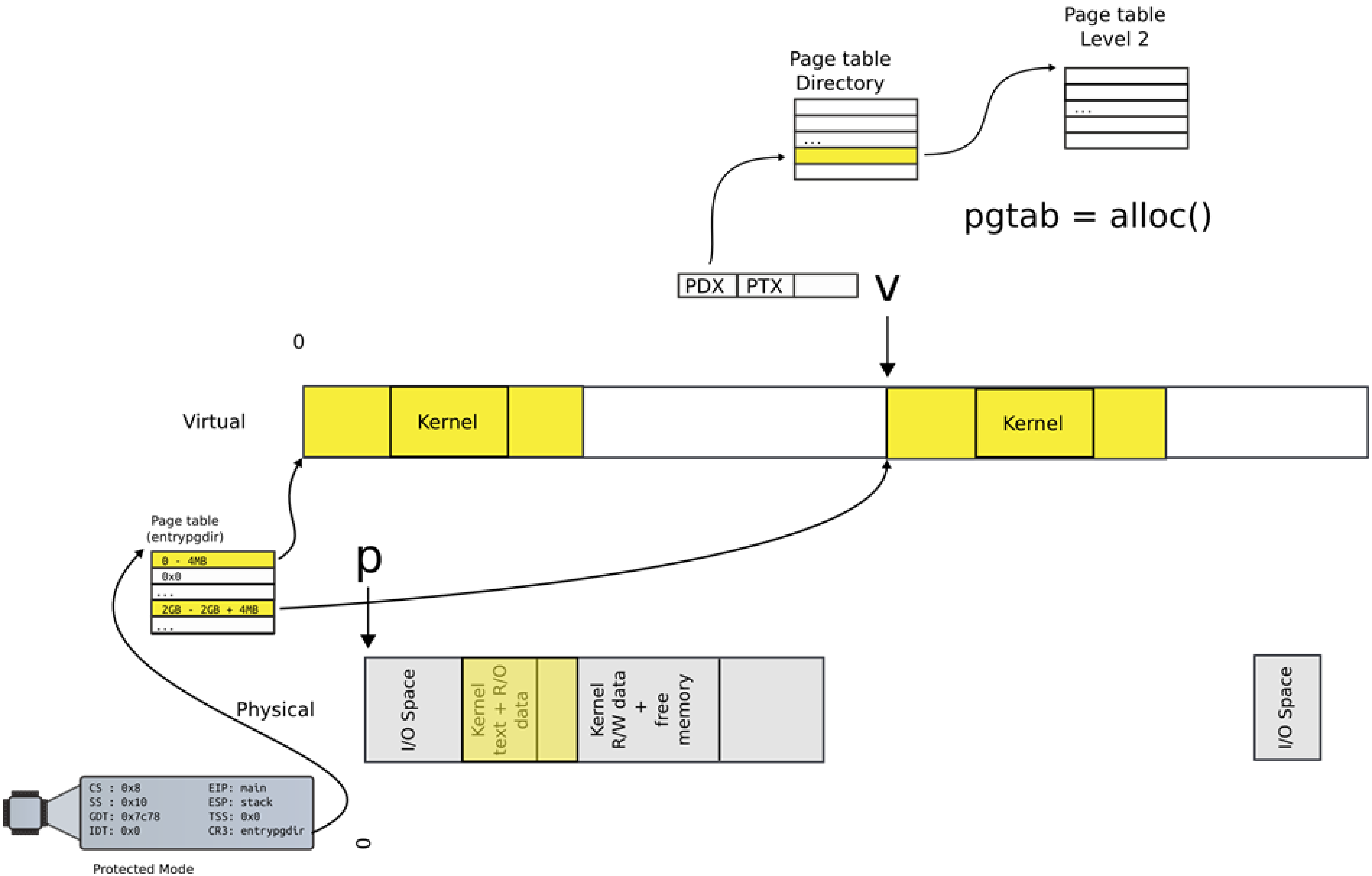
Outline

- Map a region of virtual memory into page tables
  - Start from 2GBs
  - Iterate memory page by page
  - Allocate page table directory and page tables as we go
  - Fill in page table entries with proper physical addresses

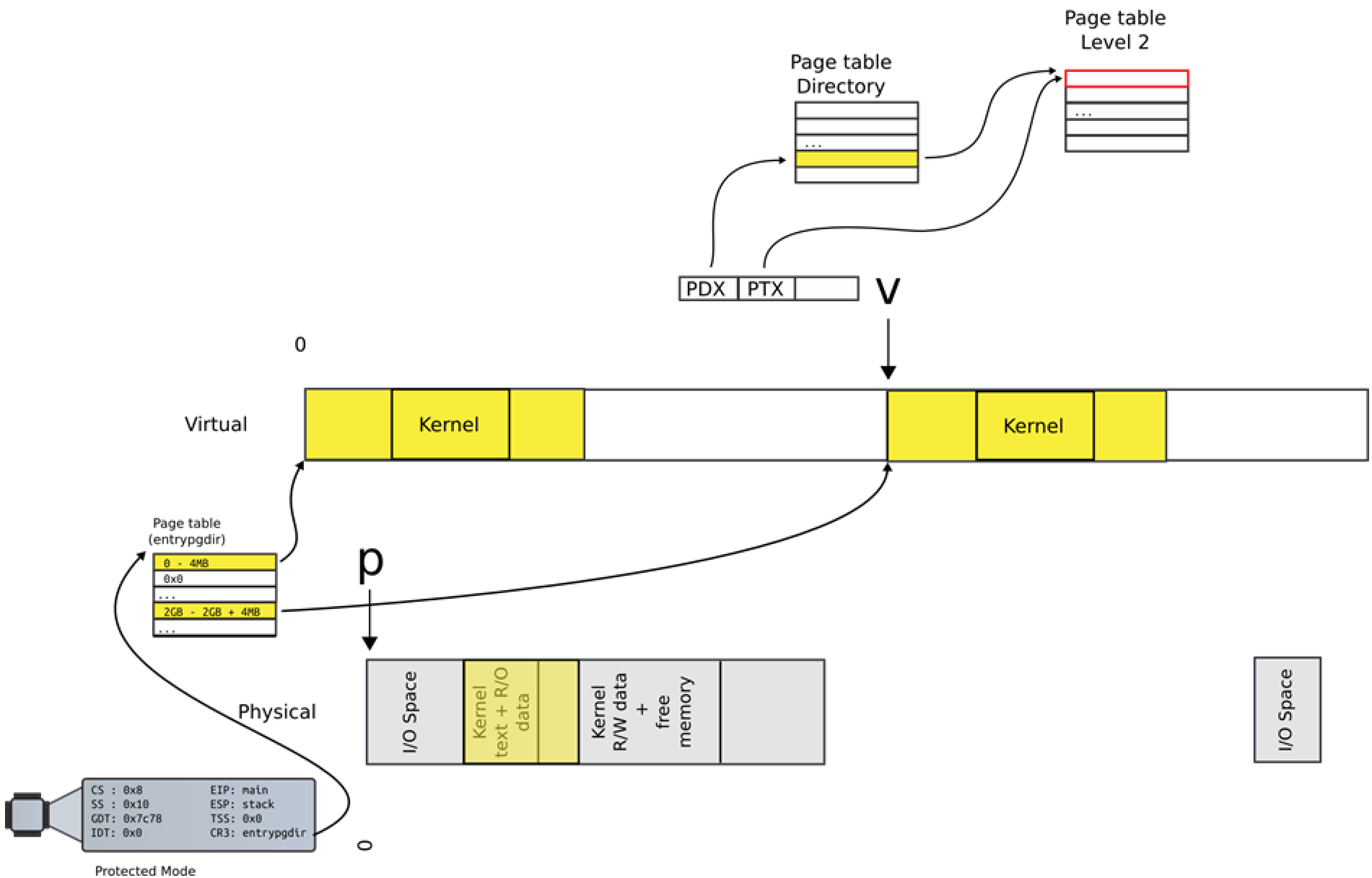
Allocate page table directory entry



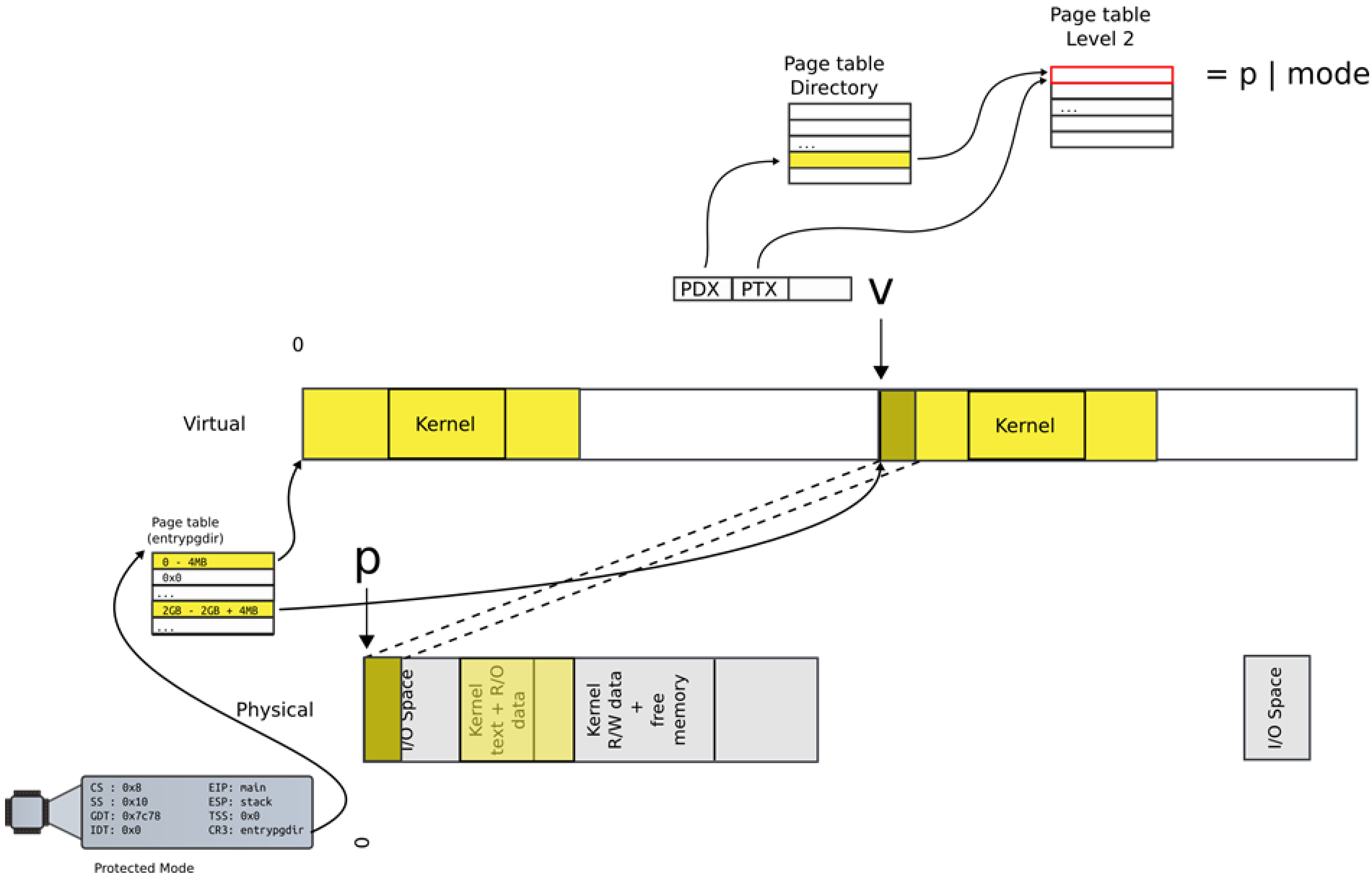
Allocate next level page table



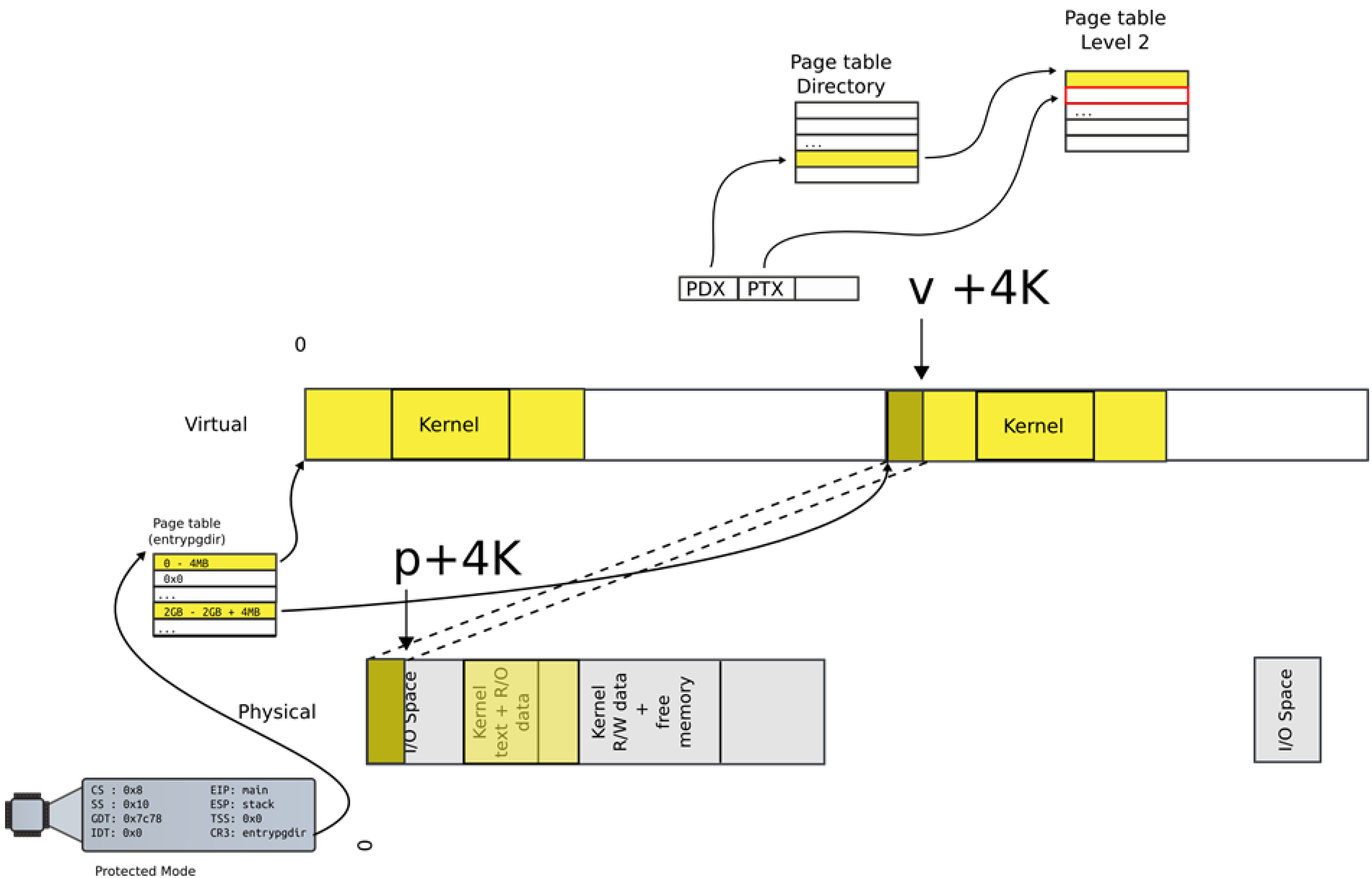
Locate PTE entry



Update mapping with physical addr



Move to next page



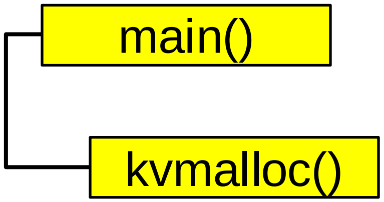
This is exactly what kernel is doing

(let’s read the source code)

Allocate page tables

```
int
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
    lapicinit(); // interrupt controller
    seginit(); // segment descriptors
    cprintf("\ncpu%d: starting xv6\n\n", cpunum());
    picinit(); // another interrupt controller
    ioapicinit(); // another interrupt controller
    consoleinit(); // console hardware
    uartinit(); // serial port
...
}
```

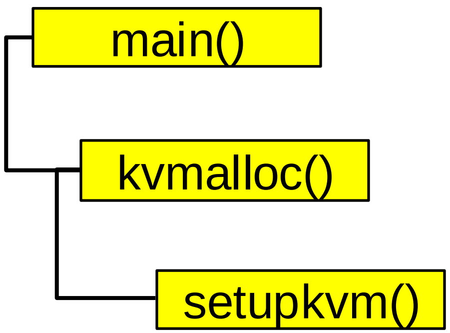
kvmalloc()




```
kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}
```

Allocate page table directory

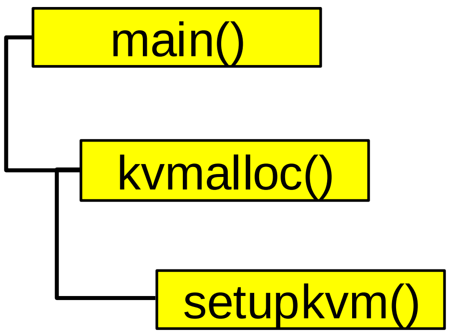
```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
...
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```



```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
...
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```

What is the address of this page?

e table
y



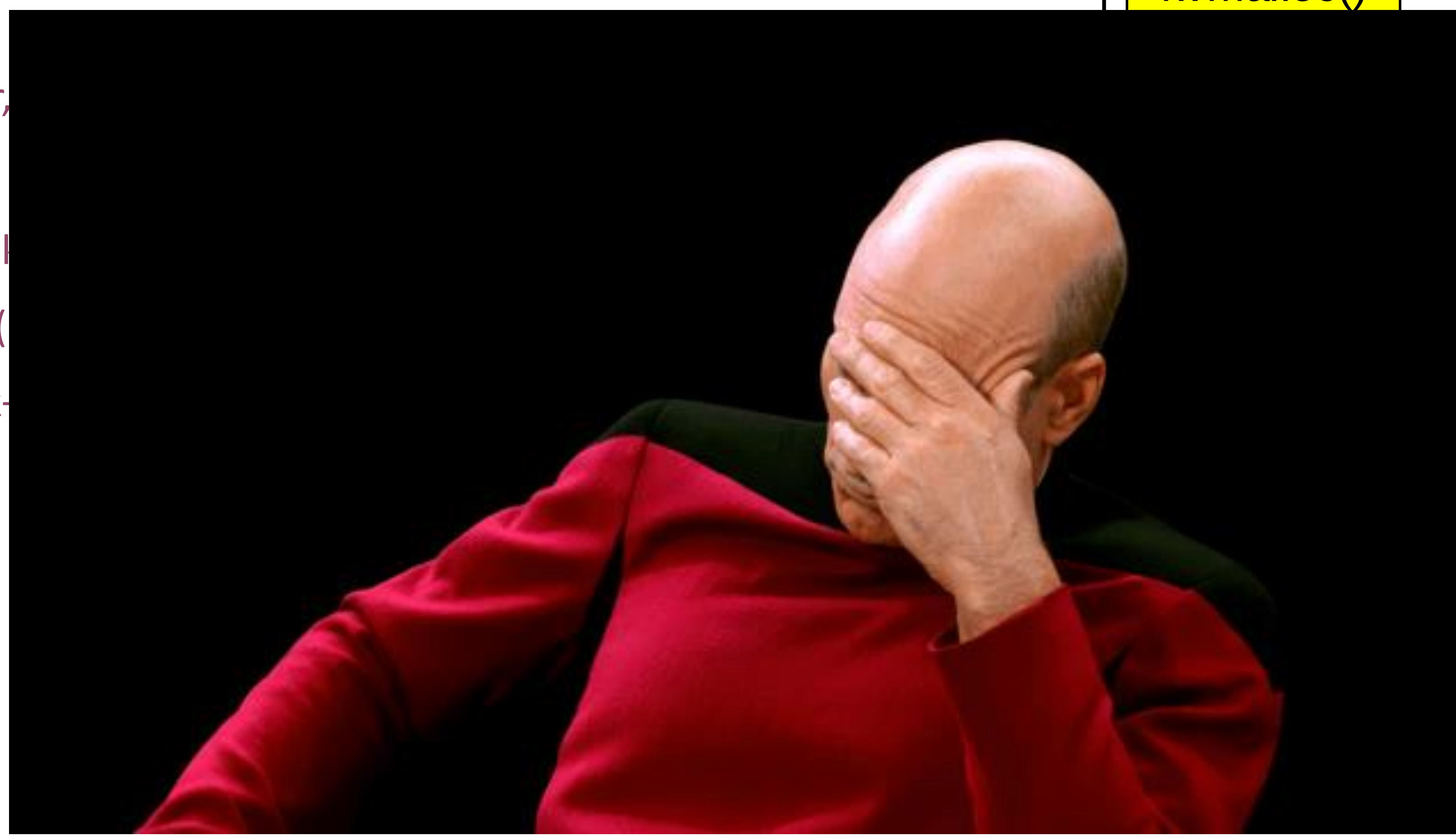
PollEv.com/antonburtsev

e table
y

What is the address of this page?

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir,
...
  for(k = kmap; k
    if(mappages(
          (uint)k-
      return 0;
  return pgdir;
}
```

main()
kvmalloc()

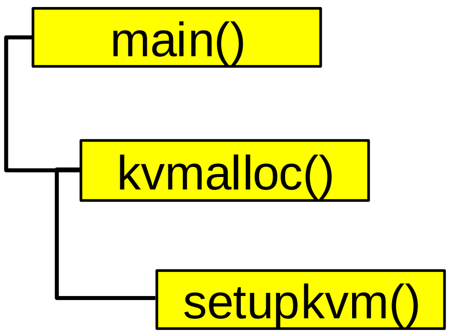


```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
...
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```

What is the address of this page?

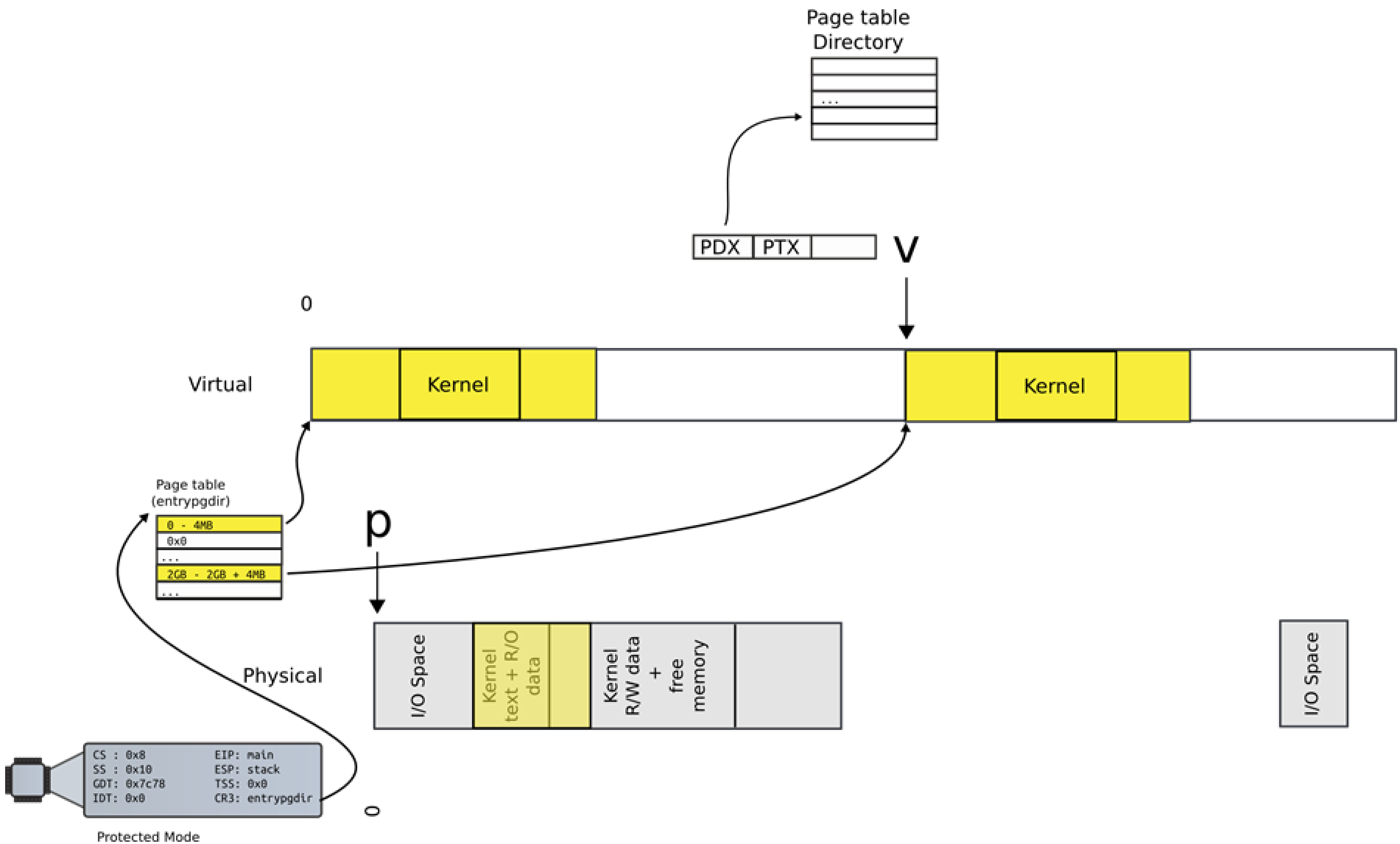
e table

y



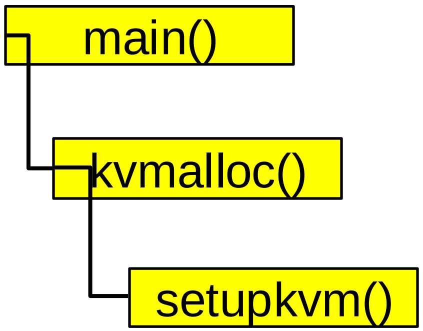
PollEv.com/antonburtsev

Allocate page table directory



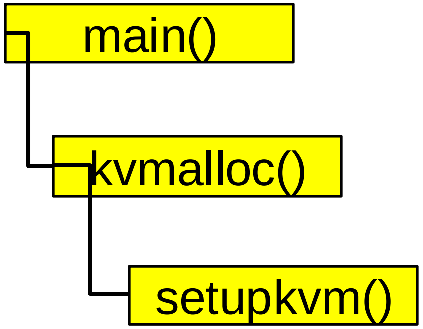
Iterate in a loop: map physical pages

```c
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
...
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```

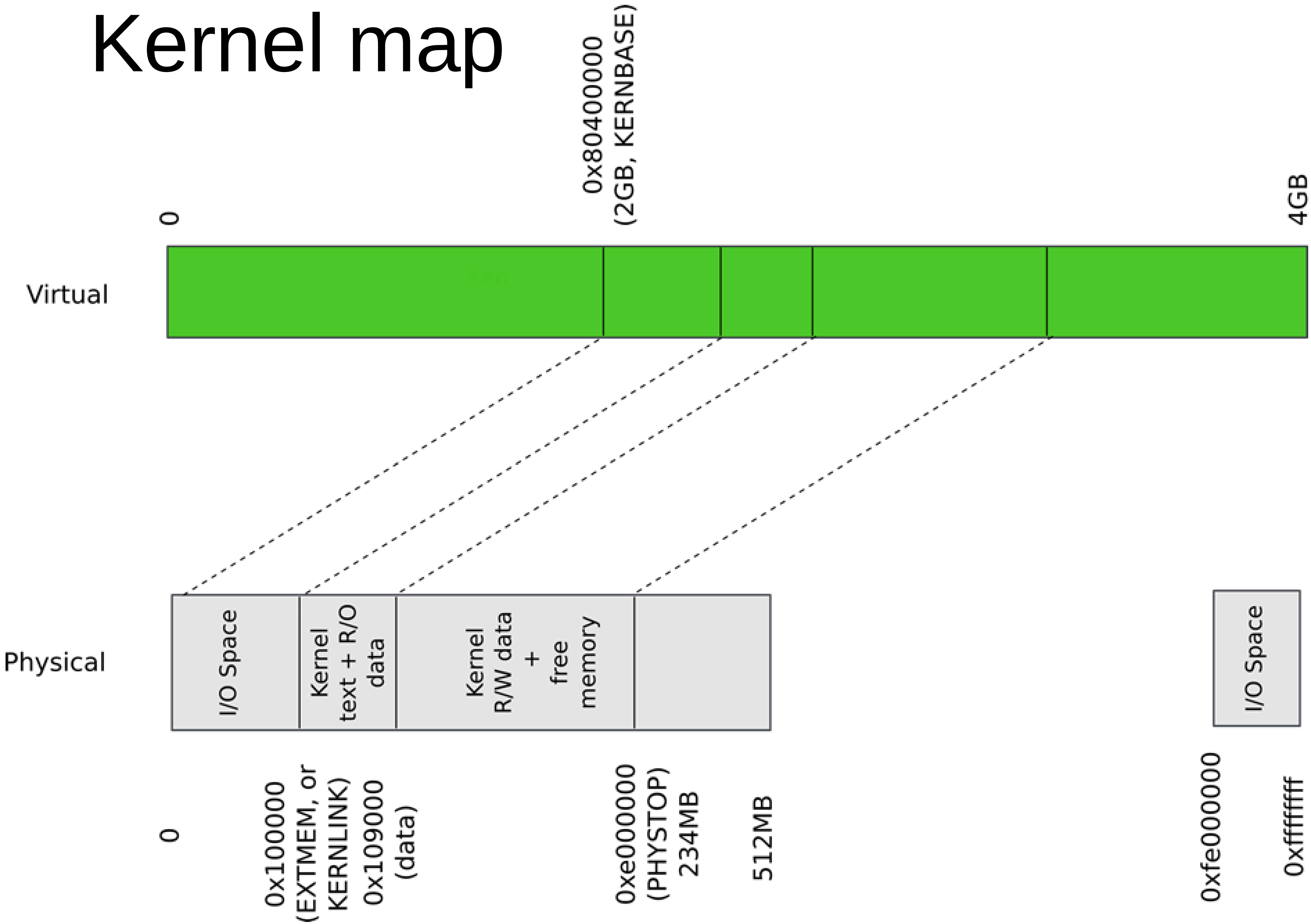


Iterate in a loop: map physical pages

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
...
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```

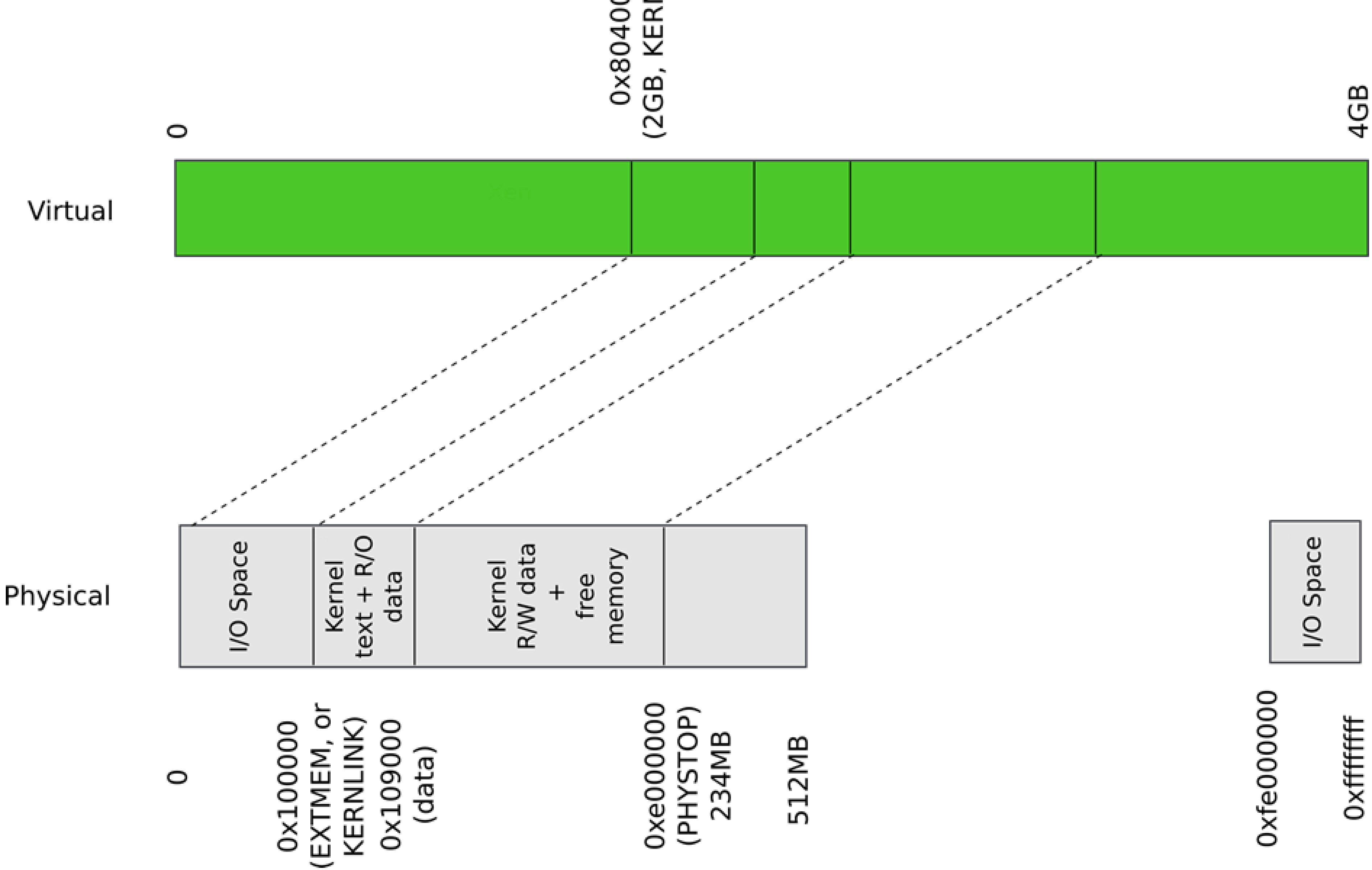


Kernel map


0
0x80400000
(2GB, KERNBASE)
4GB
Virtual
Physical
I/O Space
Kernel
text + R/O
data
Kernel
R/W data
+
free
memory
I/O Space
0
0x100000
(EXTMEM, or
KERNLINK)
0x109000
(data)
0xe000000
(PHYSTOP)
234MB
512MB
0xfe000000
0xffffffff

Kernel map

PollEv.com/antonburtsev

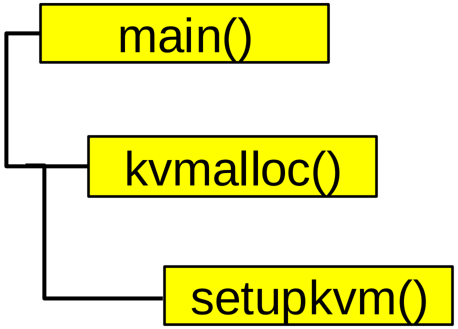


Kmap – kernel map

```
static struct kmap {
    void *virt;
    uint phys_start;
    uint phys_end;
    int perm;
} kmap[] = {
    { (void*)KERNBASE, 0, EXTMEM, PTE_W}, // I/O space
    { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},//text+rodata
    { (void*)data, V2P(data), PHYSTOP, PTE_W}, // kern data+memory
    { (void*)DEVSPACE, DEVSPACE, 0, PTE_W}, // more devices
};
```

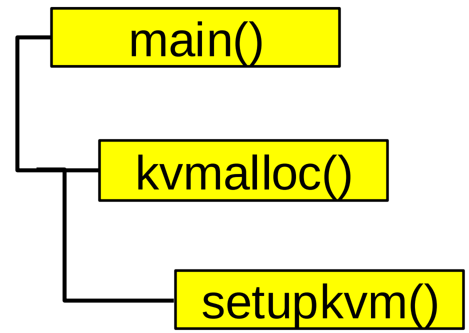
Start virtual address

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
...
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end – k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```



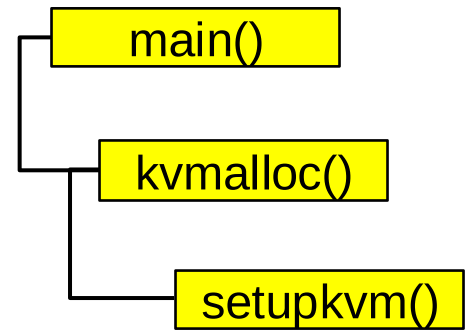
Size

```c
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
...
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```



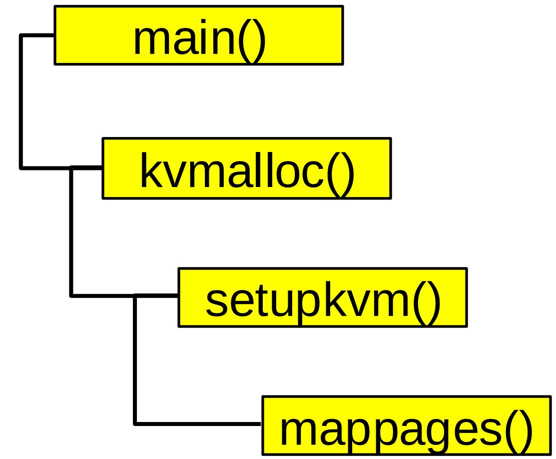
Start physical address

```c
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
...
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
              (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```



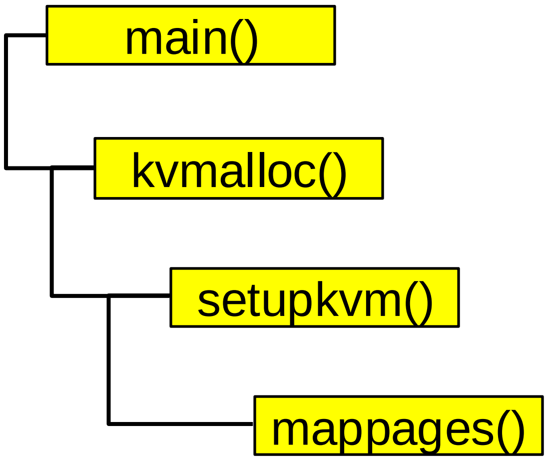
Inside mappages(

```
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size − 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return −1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```



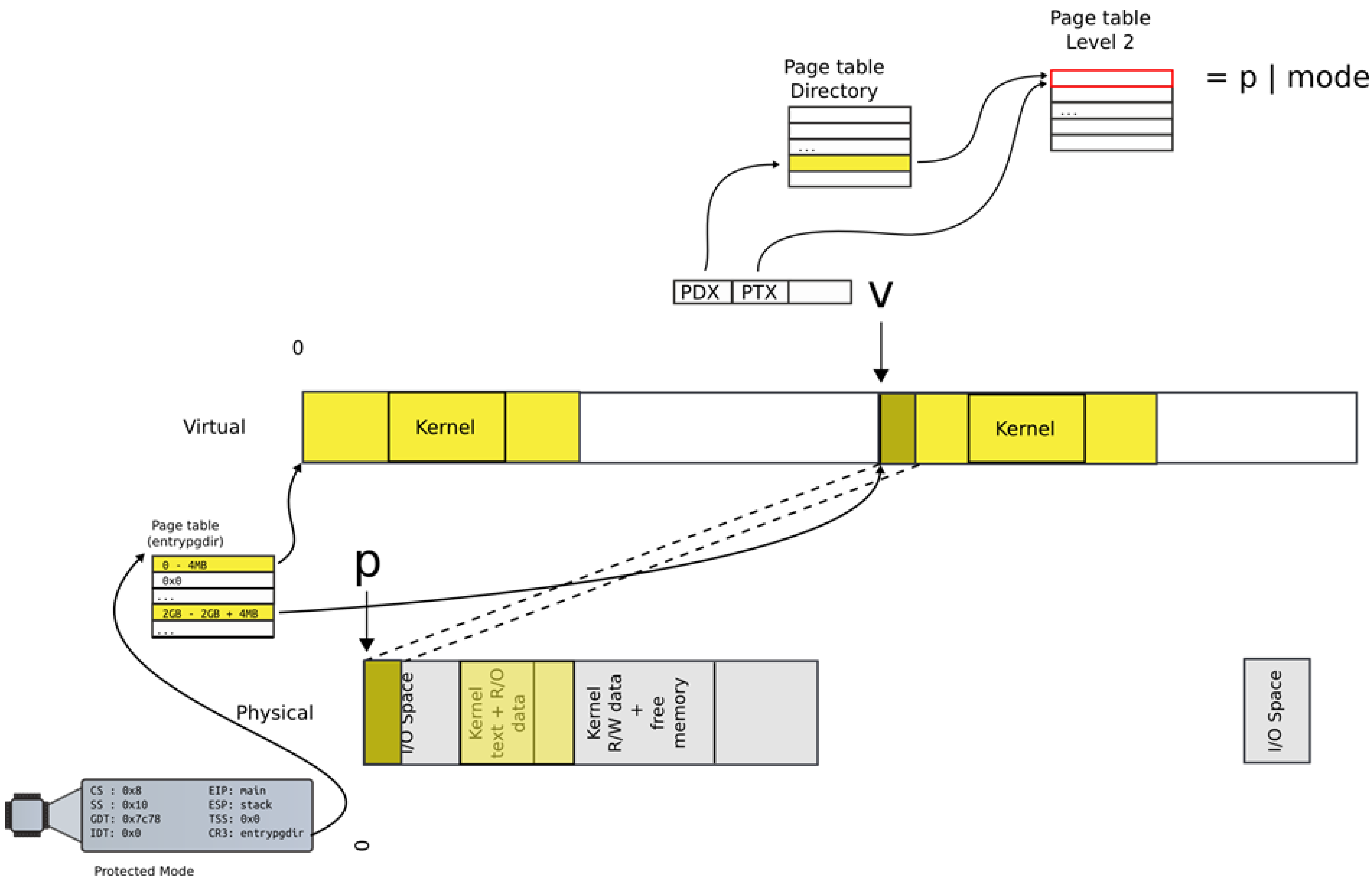
- Get the start (a) and end (last) pages fo the virtual address range we are mapping
- Then work in a loop mapping every page one by one

```c
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size − 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return −1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

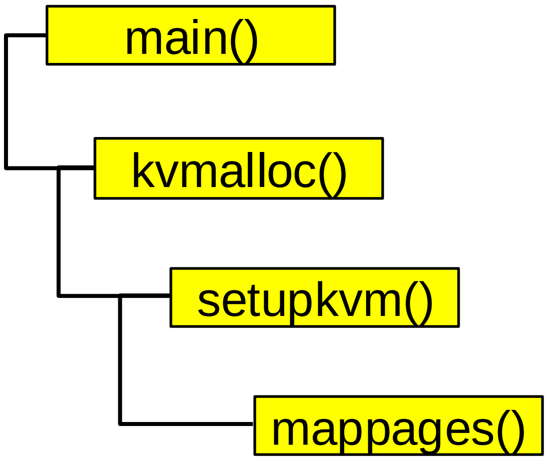


- First lookup the page table entry (pte) corresponding to the virtual address (a) we’re mapping

Locate the page table entry

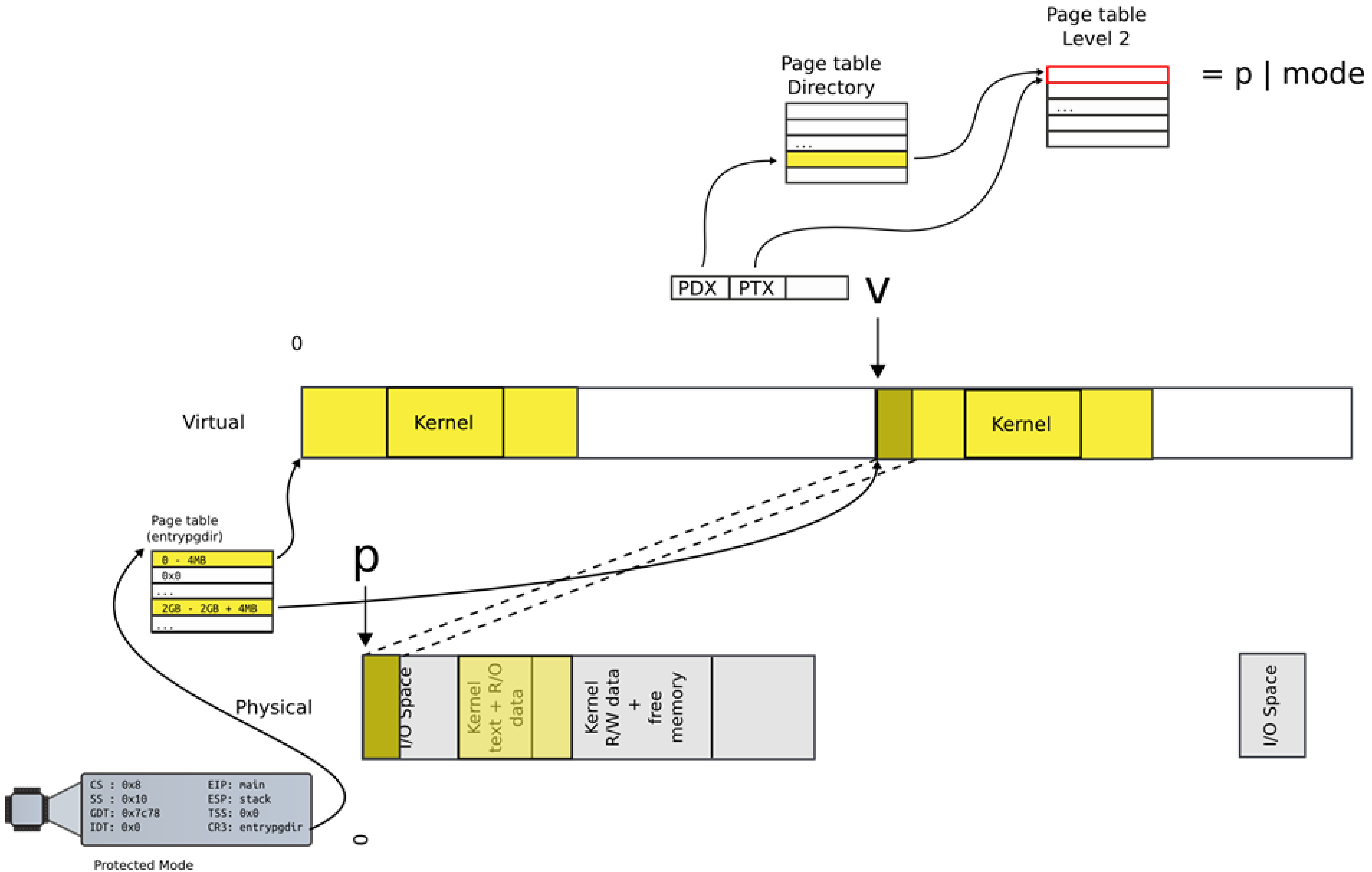


```
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size − 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return −1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

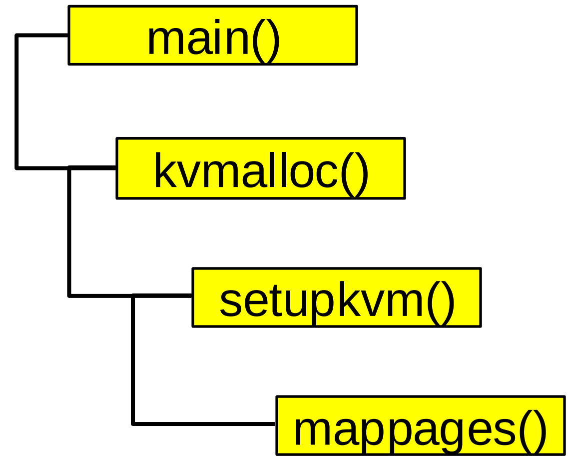


- Update the page directory entry (*pte) with the physical address (pa)

Update mapping with physical addr



```
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size − 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return −1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

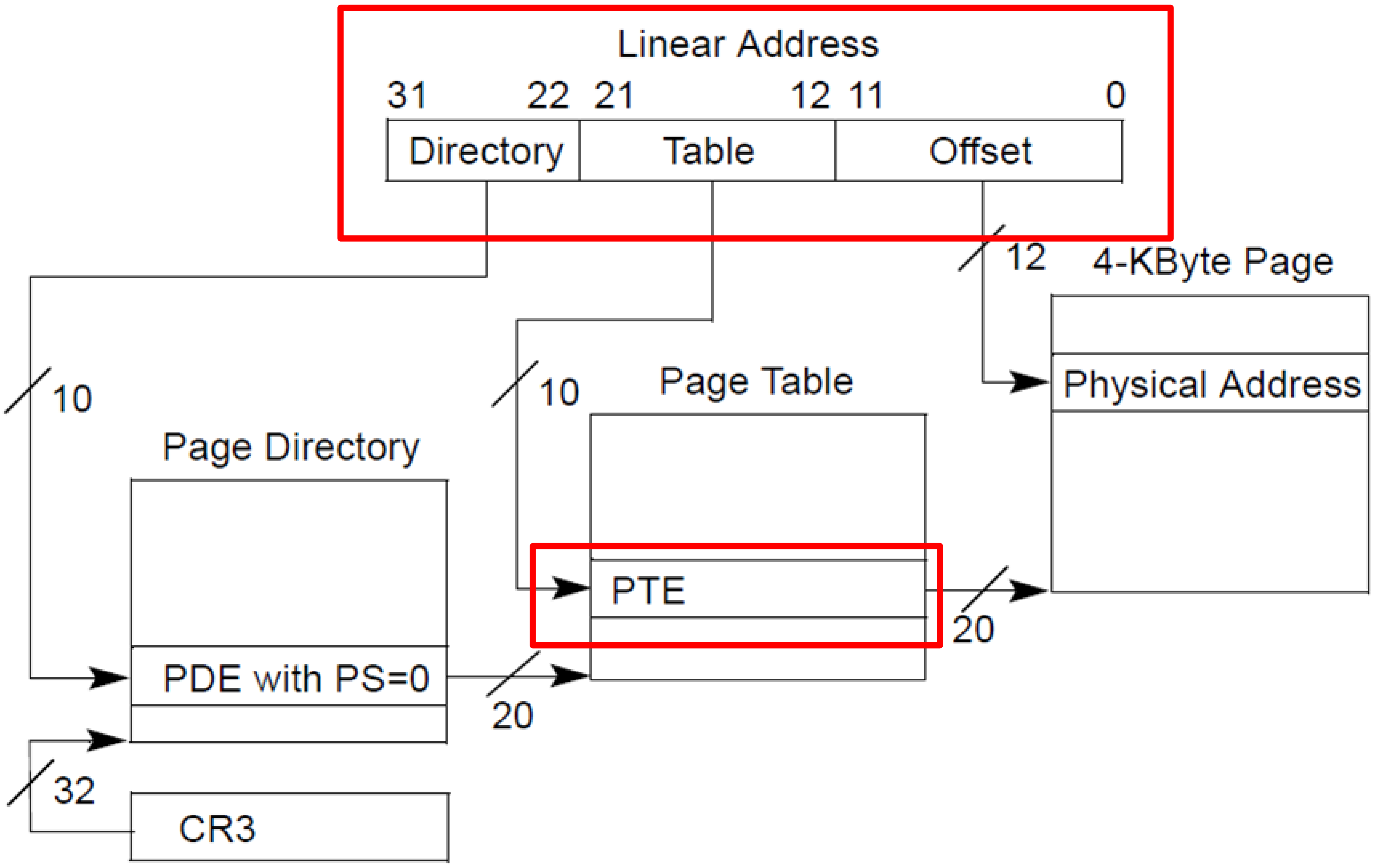


- But we need a function that locates the pte for us...

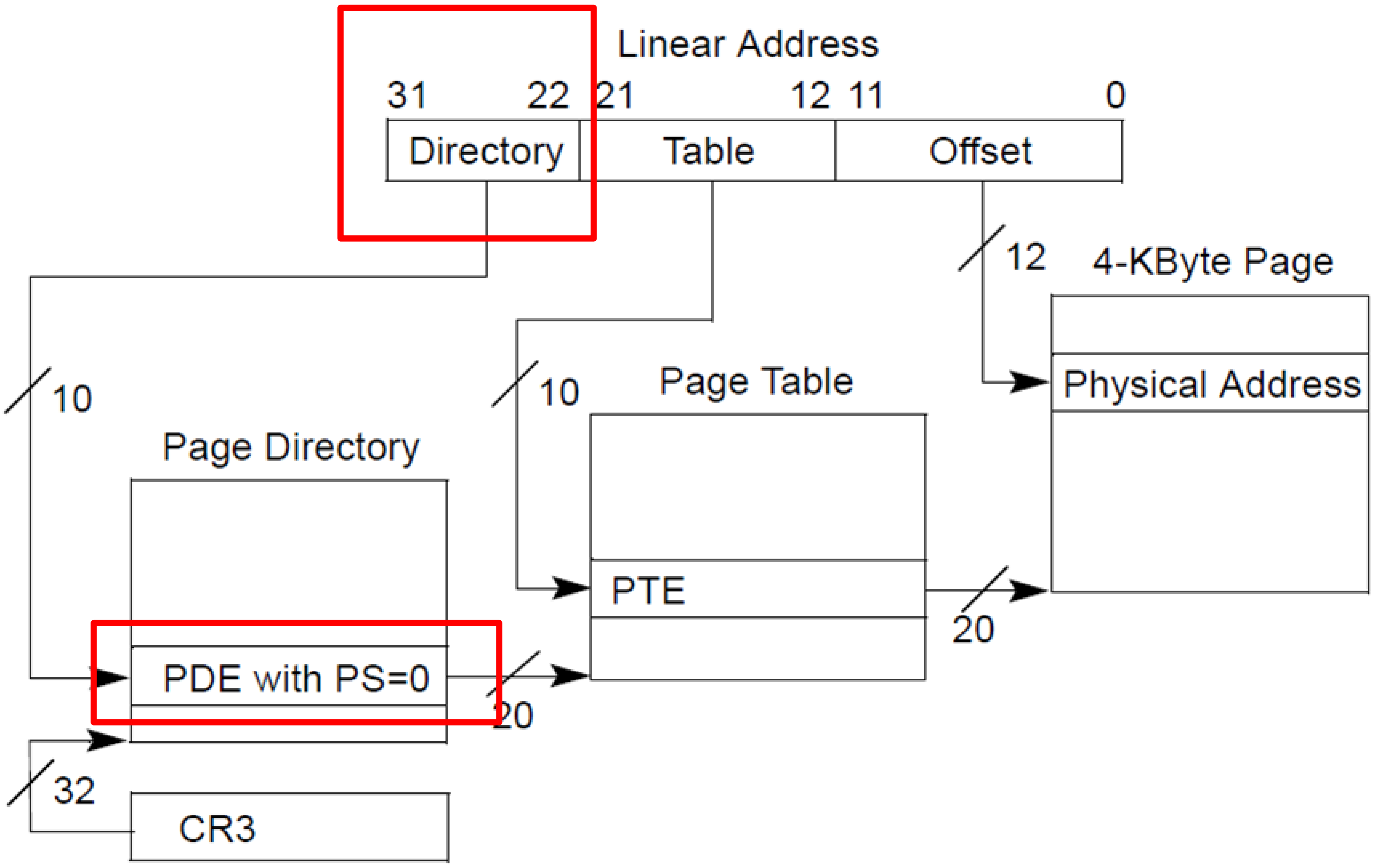
What should it look like?

- A function takes a virtual address
- Returns a page table directory entry that maps it

Recap of the page table



Locate the PDE frist



walkpgdir(): walk page table

```
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    pde_t *pde;
    pte_t *pgtab;

    pde = &pgdir[PDX(va)];
    if(*pde & PTE_P){
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
    } else {
        if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
            return 0;
        // Make sure all those PTE_P bits are zero.
        memset(pgtab, 0, PGSIZE);
...
        *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
    }
    return &pgtab[PTX(va)];
}
```

- Locate the page directory entry (*pde)

PDX()

```
// +--------10------+-------10-------+---------12----------+
// | Page Directory |    Page Table  | Offset within Page  |
// |    Index       |     Index      |                     |
// +----------------+----------------+---------------------+
// \--- PDX(va) --/ \--- PTX(va) --/

// page directory index
#define PDX(va) (((uint)(va) >> PDXSHIFT) & 0x3FF)

// page table index
#define PTX(va) (((uint)(va) >> PTXSHIFT) & 0x3FF)
...
#define PTXSHIFT 12 // offset of PTX in a linear address
#define PDXSHIFT 22 // offset of PDX in a linear address
```

PDX()

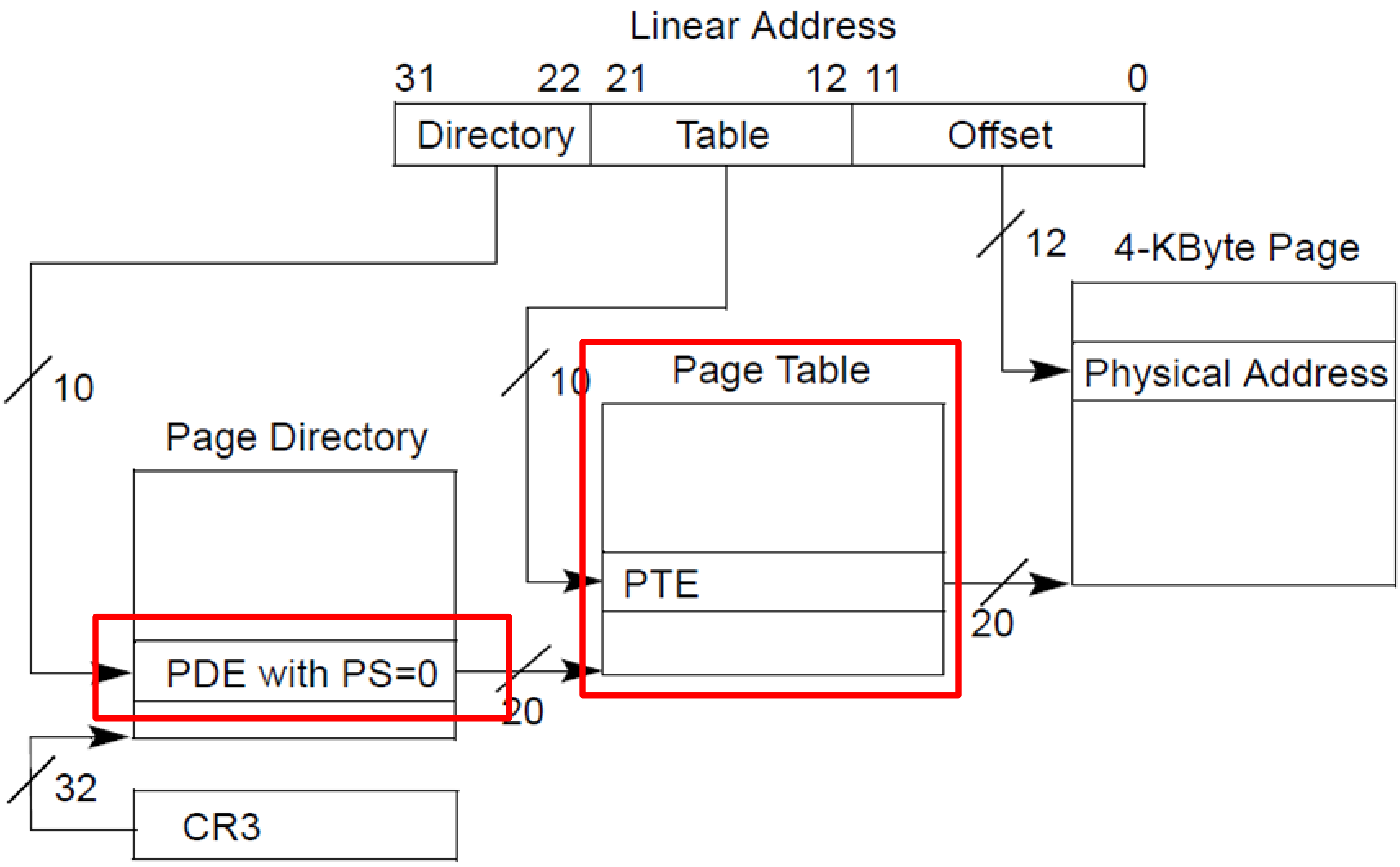
```
// +--------10------+-------10-------+---------12----------+
// | Page Directory |    Page Table   | Offset within Page  |
// |     Index      |      Index      |                     |
// +----------------+-----------------+---------------------+
// \--- PDX(va) --/ \--- PTX(va) --/

// page directory index
#define PDX(va) (((uint)(va) >> PDXSHIFT) & 0x3FF)

// page table index
#define PTX(va) (((uint)(va) >> PTXSHIFT) & 0x3FF)
...
#define PTXSHIFT 12 // offset of PTX in a linear address
#define PDXSHIFT 22 // offset of PDX in a linear address
```

walkpgdir(): walk page table

```
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    pde_t *pde;
    pte_t *pgtab;

    pde = &pgdir[PDX(va)];
    if(*pde & PTE_P){
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
    } else {
        if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
            return 0;
        // Make sure all those PTE_P bits are zero.
        memset(pgtab, 0, PGSIZE);
...
        *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
    }
    return &pgtab[PTX(va)];
}
```

- Check if page table is allocated (present)

Check if level 2 page table is allocated

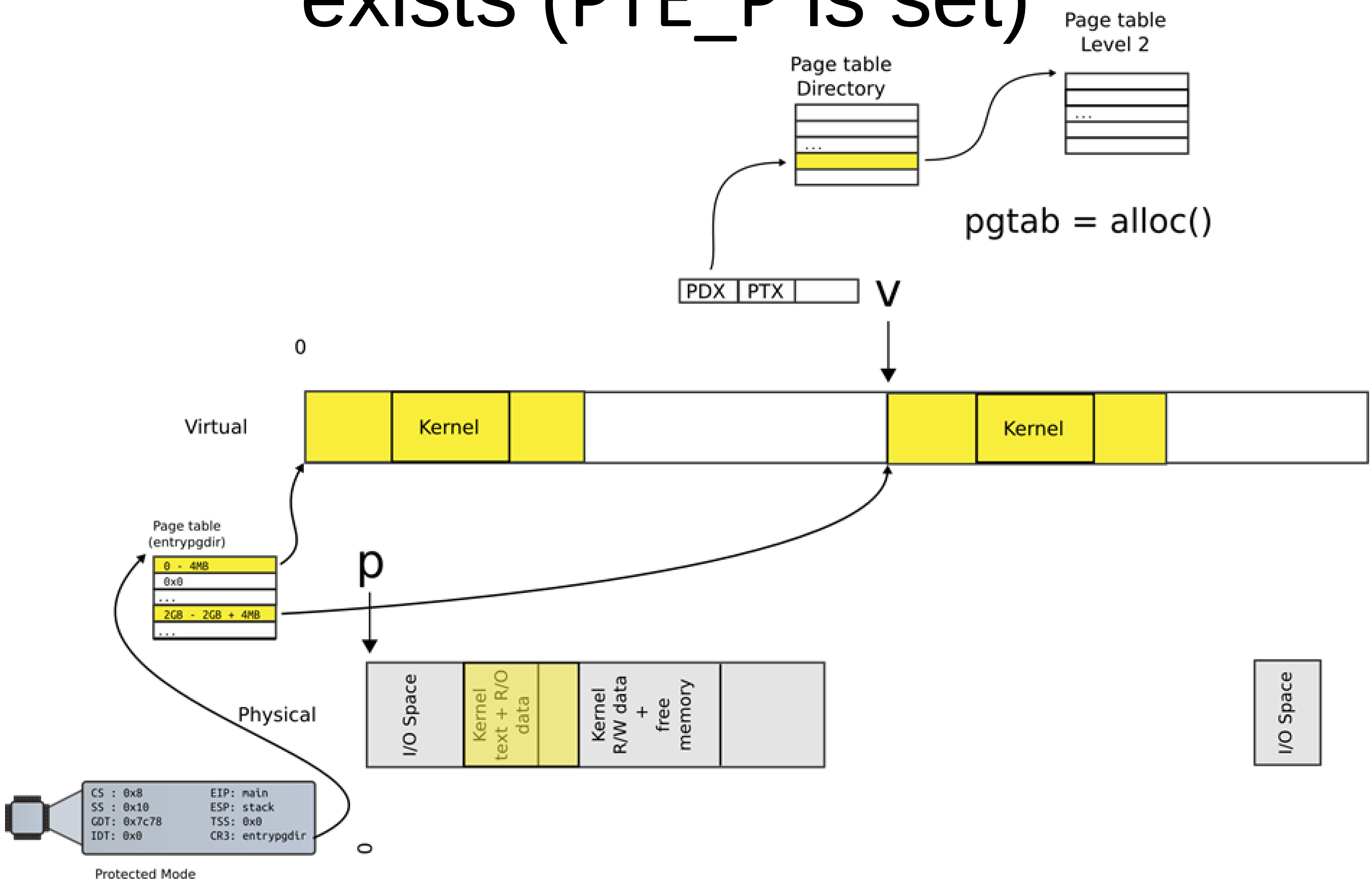


walkpgdir(): walk page table

```
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    pde_t *pde;
    pte_t *pgtab;

    pde = &pgdir[PDX(va)];
    if(*pde & PTE_P){
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
    } else {
        if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
            return 0;
        // Make sure all those PTE_P bits are zero.
        memset(pgtab, 0, PGSIZE);
...
        *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
    }
    return &pgtab[PTX(va)];
}
```

- Allocate if needed

See if the next page table level exists (`PTE_P` is set)

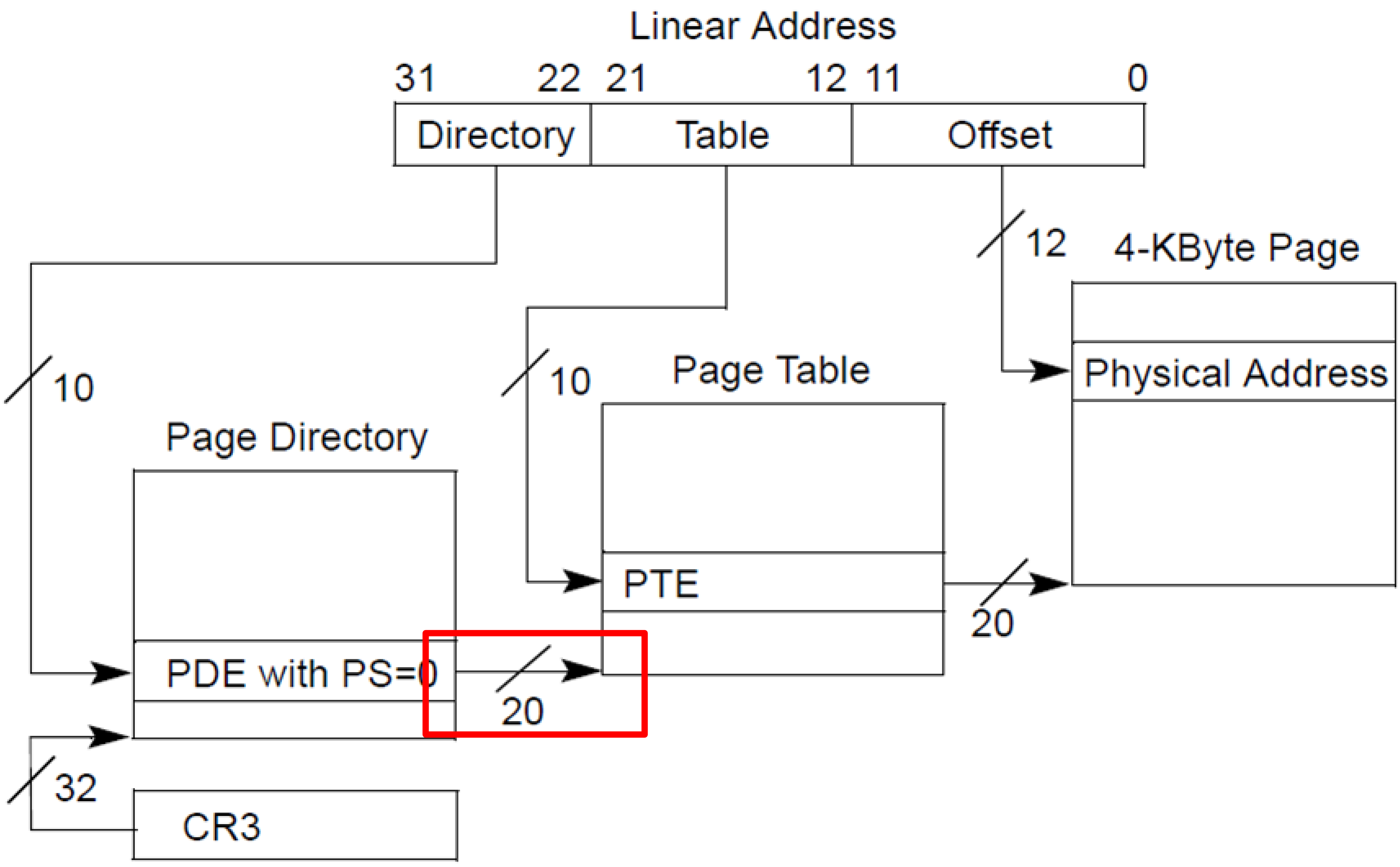


walkpgdir(): walk page table

```
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    pde_t *pde;
    pte_t *pgtab;

    pde = &pgdir[PDX(va)];
    if(*pde & PTE_P){
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
    } else {
        if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
            return 0;
        // Make sure all those PTE_P bits are zero.
        memset(pgtab, 0, PGSIZE);
...
        *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
    }
    return &pgtab[PTX(va)];
}
```

- If exists, get the address of the next level

PDE contains 20 bits which represent physical page number



Getting level 2 page

```
1761    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
```

- We need two things
- Convert from 20 bits of physical page number to physical address of the page
- `PTE_ADDR(*pde)`
- Convert from physical address of that page to virtual address
- `P2V(...)`
  - We can't access physical addresses directly
  - We can only access virtual addresses
  - Registers, mov instructions, etc. contain virtual addresses
  - Physical address have to be mapped by the current page table

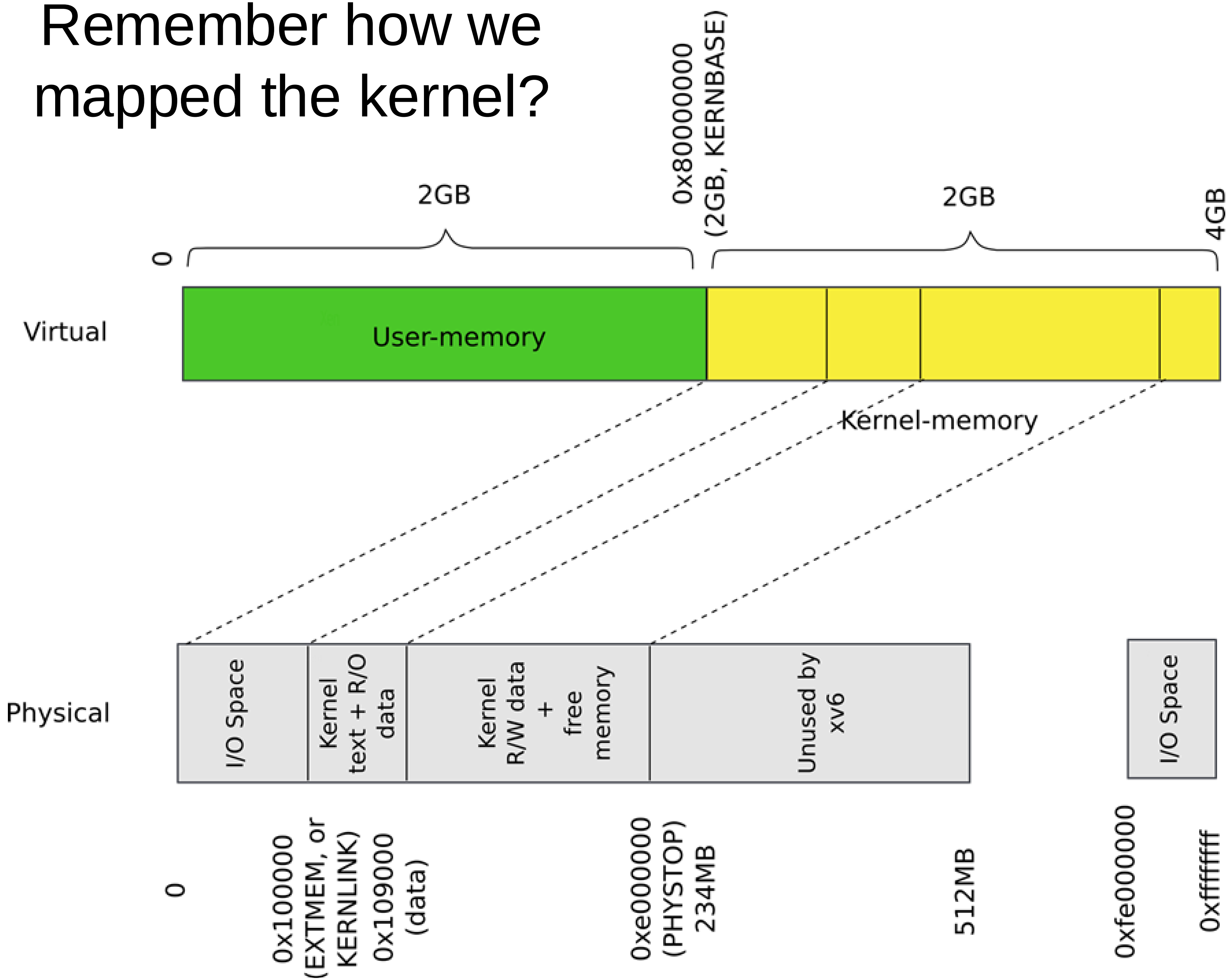
Step 1

- Convert from 20 bits of physical page number to physical address of the page
- PTE_ADDR(*pde)
- This is trivial

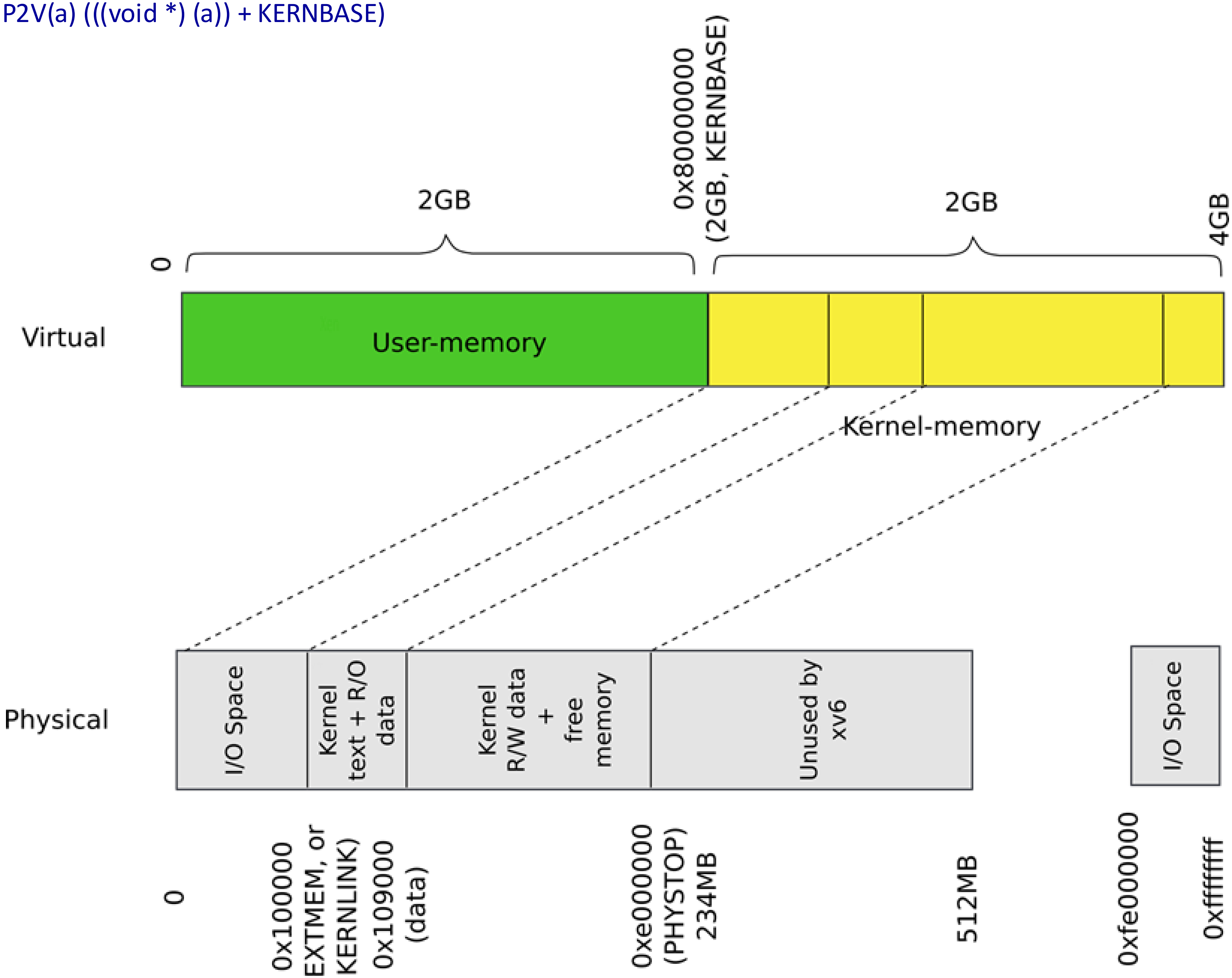
Step 2

- Convert from physical address of that page to virtual address
- P2V(...)
- This seems a bit tricky

Remember how we mapped the kernel?



```
0207 #define KERNBASE 0x80000000 // First kernel virtual address

0210 #define V2P(a) (((uint) (a)) – KERNBASE)

0211 #define P2V(a) (((void *) (a)) + KERNBASE)
```

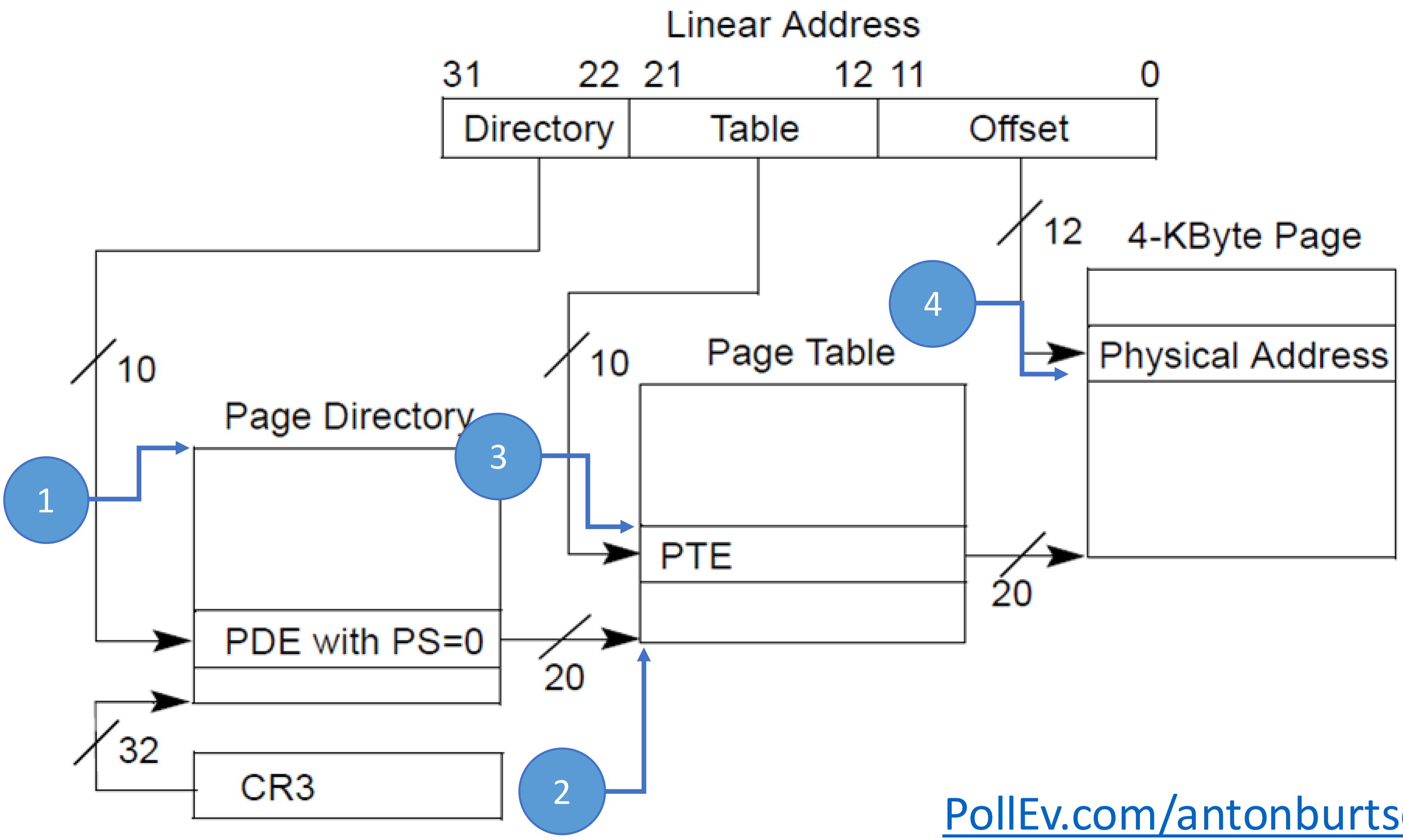


walkpgdir(): walk page table

```
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    pde_t *pde;
    pte_t *pgtab;

    pde = &pgdir[PDX(va)];
    if(*pde & PTE_P){
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
    } else {
        if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
            return 0;
        // Make sure all those PTE_P bits are zero.
        memset(pgtab, 0, PGSIZE);
...
        *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
    }
    return &pgtab[PTX(va)];
}
```

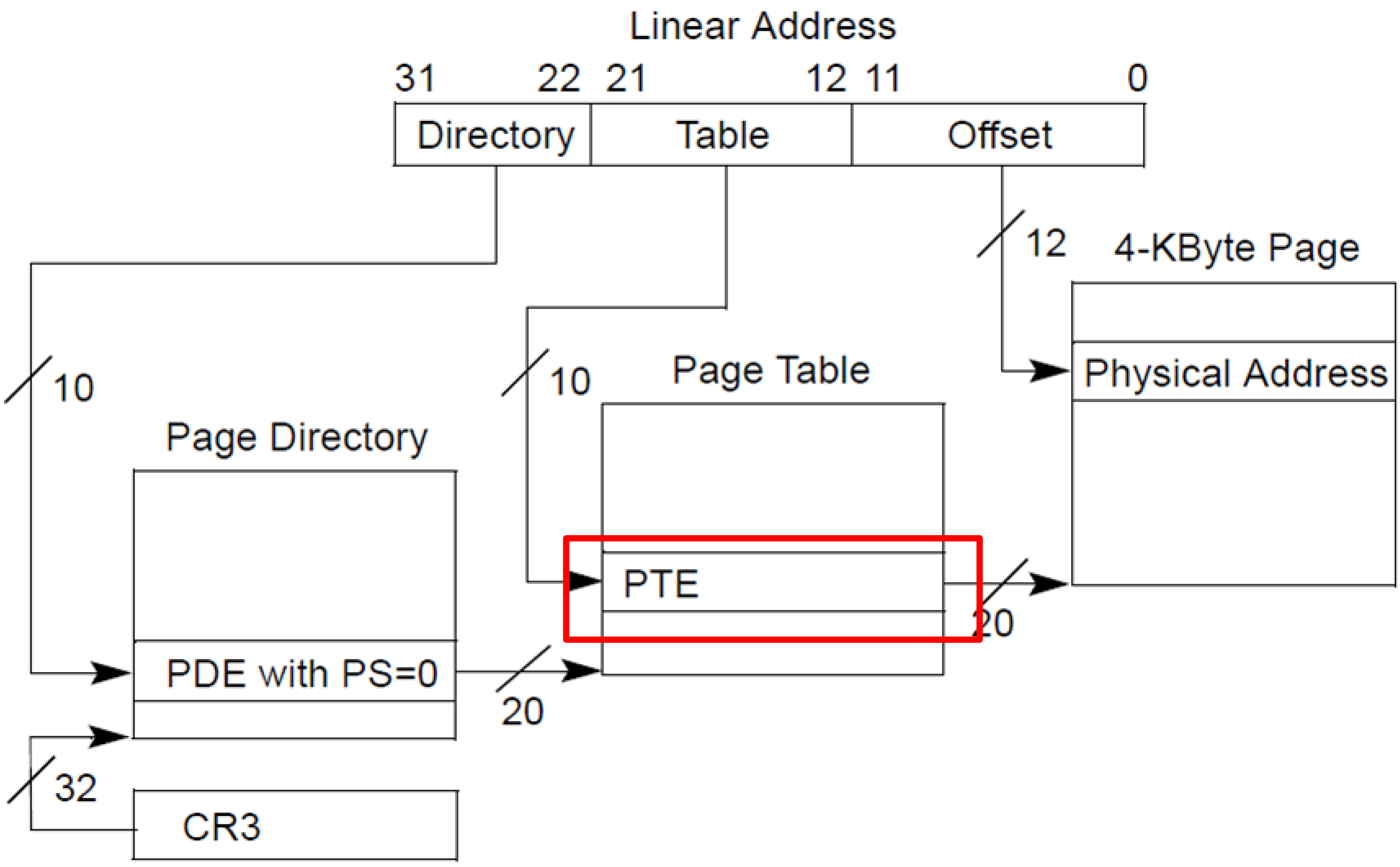
- Return pointer to the PTE

Poll: what are we returning?



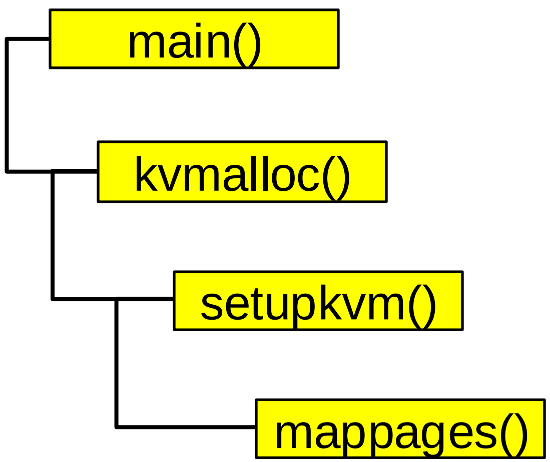
PollEv.com/antonburtsev

Return a pointer to PTE



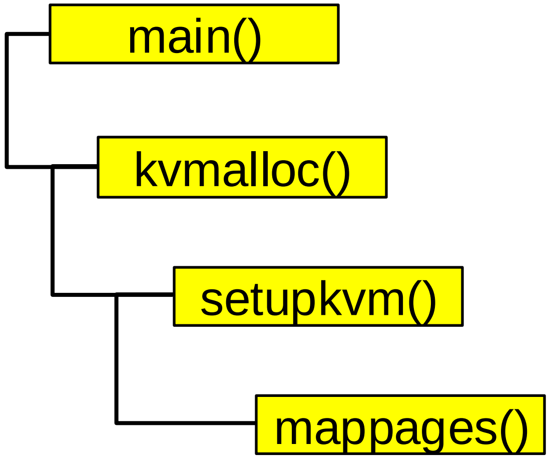
Back to mappages() function that maps a region of virtual memory into continuous region of physical memory

```c
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size − 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return −1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```



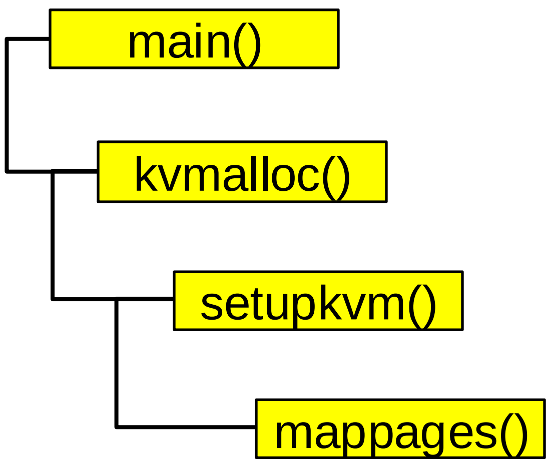
Remember we just discussed walkpgdir()

```c
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size − 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return −1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```



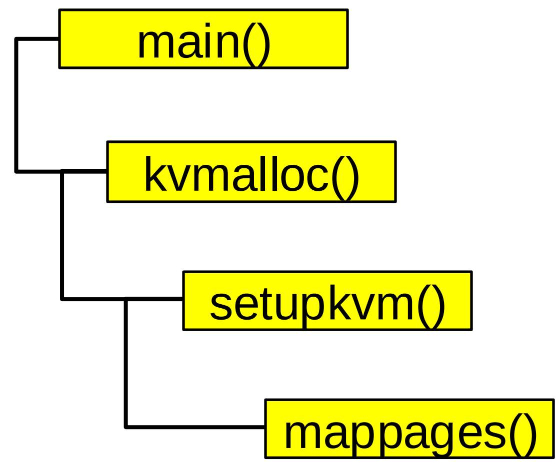
Page present (PTE_P) – panic

```
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size − 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return −1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```



- Update page table entry
- Where does *pte point?
  - pa – physical address of the page

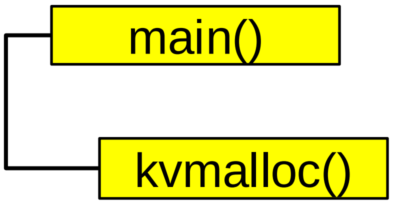
```
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size − 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return −1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```



- Move to the next page

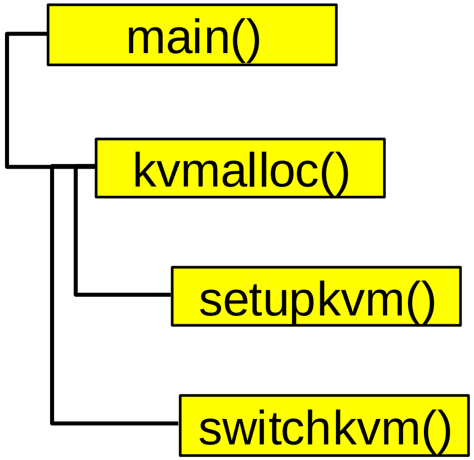
kvmalloc()

```c
kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}
```



Switch to the new page table

```
void
switchkvm(void)
{
  lcr3(v2p(kpgdir));
}
```



Recap

- Kernel has a memory allocator
- Kernel has a its own address space
- It uses 4KB page tables
- It is ready to create processes

```
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc(); // kernel page table
  mpinit(); // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  cprintf("\ncpu%d: starting xv6\n\n", cpunum());
  picinit();       // another interrupt controller
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  if(!ismp)
    timerinit();   // uniprocessor timer
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
```

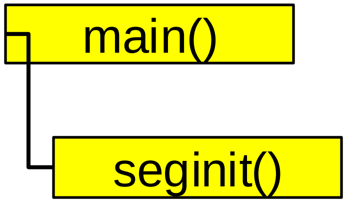
main()

```
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc(); // kernel page table
  mpinit(); // detect other processors
  lapicinit();    // interrupt controller
  seginit();      // segment descriptors
  cprintf("\ncpu%d: starting xv6\n\n", cpunum());
  picinit();      // another interrupt controller
  ioapicinit();   // another interrupt controller
  consoleinit();  // console hardware
  uartinit();     // serial port
  pinit();        // process table
  tvinit();       // trap vectors
  binit();        // buffer cache
  fileinit();     // file table
  ideinit();      // disk
  if(!ismp)
    timerinit();  // uniprocessor timer
  startothers();  // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();     // first user process
  mpmain();       // finish this processor's setup
}
```

main()

Initialize GDT

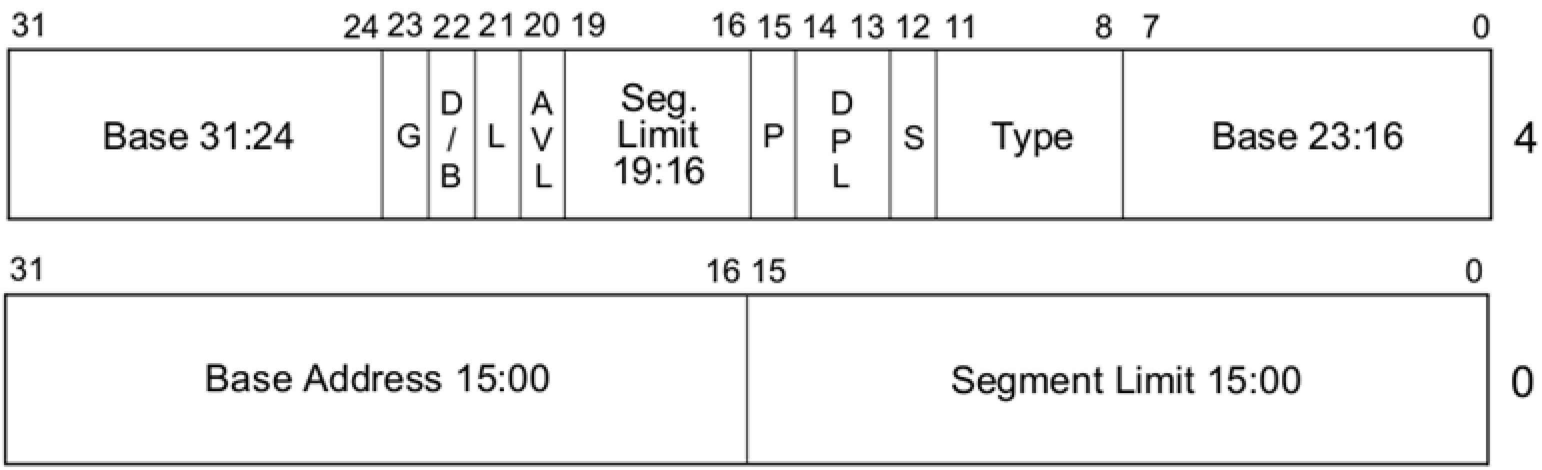
```
// Set up CPU's kernel segment descriptors.
// Run once on entry on each CPU.
void
seginit(void)
{
  struct cpu *c;

  // Map "logical" addresses to virtual addresses using identity map.
  // Cannot share a CODE descriptor for both kernel and user
  // because it would have to have DPL_USR, but the CPU forbids
  // an interrupt from CPL=0 to DPL=3.
  c = &cpus[cpuid()];
  c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
  c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
  lgdt(c->gdt, sizeof(c->gdt));
}
```



Struct CPU

```
// Per−CPU state
struct cpu {
  uchar apicid;              // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                  // Depth of pushcli nesting.
  int intena;                // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null
};

extern struct cpu cpus[NCPU];
```

Segment descriptor



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Segment Descriptor

```
// Segment Descriptor
struct segdesc {
  uint lim_15_0 : 16;  // Low bits of segment limit
  uint base_15_0 : 16; // Low bits of segment base address
  uint base_23_16 : 8; // Middle bits of segment base address
  uint type : 4;       // Segment type (see STS_ constants)
  uint s : 1;          // 0 = system, 1 = application
  uint dpl : 2;        // Descriptor Privilege Level
  uint p : 1;          // Present
  uint lim_19_16 : 4;  // High bits of segment limit
  uint avl : 1;        // Unused (available for software use)
  uint rsv1 : 1;       // Reserved
  uint db : 1;         // 0 = 16−bit segment, 1 = 32−bit segment
  uint g : 1;          // Granularity: limit scaled by 4K when set
  uint base_31_24 : 8; // High bits of segment base address
};
```

Thank you!

(Next time: interrupts!)